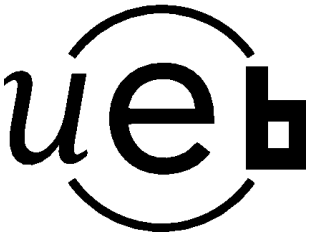
ANNÉE 2013



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

François FOUQUET

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires ISTIC

Kevoree : Model@Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes

Thèse soutenue à Rennes
le 6 Mars 2013

devant le jury composé de :

François TAÏANI
Professeur, Université de Rennes 1 / *Président*

Rachid GUERRAOUI
Professeur, School of Computer and Communication Sciences / *Rapporteur*

Lionel SEINTURIER
Professeur, Université de Lille 1 / *Rapporteur*

Vivien QUEMA
Professeur, Université de Grenoble 1, / *Examinateur*

Yves LE TRAON
Professeur, University of Luxembourg / *Examinateur*

Jean-Marc JÉZÉQUEL
Professeur, Université de Rennes 1 / *Directeur de thèse*

Noël PLOUZEAU
Maître de conférences, Université de Rennes 1 / *Co-directeur de thèse*

Olivier BARAIS
Maître de conférences, Université de Rennes 1 / *Co-directeur de thèse*

Remerciements

Quelle étrange tradition que de commencer par la fin ! Néanmoins j'espère que ces remerciements et cette fin de doctorat ne sont que le commencement pour moi de nombreux autres travaux de recherches. En effet, je sors de cette thèse plus passionné que jamais par l'étude de ces grands systèmes informatiques qui révolutionne notre façon de vivre. Cette thèse, ces travaux et ces résultats n'ont pus être possibles que grâce à la participation de nombreuses personnes.

Je tiens tout d'abord à remercier les membres du jury dont j'ai apprécié la lecture attentive et dont les remarques ont permis l'amélioration de ce document. Mais au delà, je souhaite les remercier, pour avoir été une source d'inspiration, chacun dans des aspects différents de ces résultats.

Je voudrais également remercier tout particulièrement Jean-Marc Jézéquel, Noël Plouzeau et Olivier Barais. Eux trois que j'ai eu le plaisir de côtoyer lors de mon cursus de Master ont su me faire découvrir et me donner le goût de la recherche, tout en m'aidant à construire les connaissances fondamentales pour celle-ci. Ce sont eux également qui m'ont offert l'opportunité de mener à bien ce doctorat au sein d'une équipe et avec un sujet passionnant à tout point de vue.

Je tiens de fait à saluer et remercier tous les membres passés et présents de l'équipe Triskell. Collègues et amis, c'est aux travers de nos discussions, innombrables cafés, prototypes et implantations que se sont forgées les idées de cette thèse. Merci tout particulièrement à Erwan Daubert, Gregory Nain, Brice Morin et Jean-Emile Dartois pour leur aide précieuse lors du développement de Kevoree mais au delà pour leur soutien moral si appréciable durant ces trois ans. Merci également à Didier Vojtisek pour son important travail de coordination ainsi que Loic Lesage pour tous les aspects d'organisation. De plus, je tiens à remercier Benoît Baudry, pour son aide de synthèse mais surtout pour nos discussions qui avec Johan Bourcier, m'amène aujourd'hui à considérer les systèmes informatiques de façon plus "biologique" !

Il est également important pour moi de remercier l'équipe d'Arnor Solberg en Norvège ainsi que Franck Fleurey pour m'avoir non seulement accueilli, permis de pousser Kevoree dans ses retranchements mais également fait découvrir un pays extraordinaire dont cette contribution tire bien plus qu'une partie de son nom...

Merci également à Sylvain Thomas et Jacques Freydrich qui sur des aspects différents m'ont aidé à construire ma vision d'architecte logiciel.

Sur un plan plus personnel, je souhaite remercier tous les membres de mon entourage qui ont su me soutenir et m'épauler durant ces trois années de travail et de doutes inhérents à la recherche. Merci donc à Jean-Luc pour m'avoir donné très tôt le goût des sciences, merci à Maryse pour m'avoir donné le goût de la communication orale et écrite et à Emilie pour son esprit de "fouille", merci à tous les trois pour avoir subi et corrigé

mes innombrables fautes d'orthographes ! Enfin merci à Clotilde pour sa patience, son soutien et son accompagnement constant durant ces trois années.

Il est temps pour moi de remercier une dernière fois tous ceux que je n'ai pas explicitement cités, et de continuer car je suis impatient de donner suite à ces travaux.

Table des matières

Chapitre 0

Résumé de thèse

0.1 Contexte : vers des systèmes distribués dynamiquement adaptables

Les outils et méthodes permettant de développer les systèmes d'informations aujourd'hui omniprésents dans notre environnement sont en constante évolution, influencés à la fois par les usages, les volumes d'informations manipulés mais surtout par la plasticité et la durée de vie de ces systèmes. Des contraintes économiques et sociales, telles que la rapidité de mise sur le marché, ont remis en cause le classique modèle de développement en V vers un cycle de plus en plus continu qui boucle sur les spécifications tout en déployant régulièrement pour récolter au plus tôt les retours des utilisateurs finaux. Dans ce contexte, l'arrêt complet et le redémarrage de ces systèmes n'est pas toujours économiquement ou fonctionnellement viable, ce qui pousse à les penser et les construire comme des systèmes adaptatifs, c'est-à-dire capables de mettre à jour leurs fonctionnalités en cours de fonctionnement. Autrefois réservés à des cas d'usages critiques comme des systèmes de surveillance ou des centraux téléphoniques, les systèmes dynamiquement adaptables (Dynamically Adaptive System (DAS) pour *Dynamically Adaptive System*) sont exploités désormais pour des projets beaucoup plus modestes comme des box internet ou des Smartphones.

À ce problème de plasticité des systèmes s'ajoute un problème de *distribution* (c.a.d répartition du système sur et pour exploiter plusieurs noeuds de calculs). En effet pendant ces cinq dernières années le nombre de terminaux mobiles a très fortement augmenté, selon une étude de Gartner [Gar10], les ventes de Smartphones augmente de 96% entre 2009 et 2010 pour attendre 80 millions d'unités vendues par trimestre. À ces architectures mobiles viennent se greffer un ensemble de matériels et capteurs embarqués regroupés sous l'appellation Internet des Objets, connectant les systèmes informatiques à leur contexte physique. Enfin le récent essor des architectures *Cloud computing* (Informatique dans les nuages) montre que la distribution massive et la mutualisation sont une réponse à la forte montée en charge. La distribution, à savoir la répartition des constituants d'un système d'information sur les différents nœuds qui le composent est donc devenue inévitable, et par extension la gestion de l'hétérogénéité de ces nœuds

également.

L'adaptation dynamique d'un système distribué pose un certain nombre de nouveaux problèmes. En effet l'adaptation d'un système est définie par Georgas *et al*[GT04] comme une réaction à un stimuli en prenant en compte le contexte d'exécution pour calculer le prochain état du DAS. La maintenance de la cohérence de cet état dans un environnement distribué reste un challenge, particulièrement dans des environnements mobiles où les communications sont intermittentes.

Directement issue de la mouvance des méthodes et outils autour de l'Ingénierie dirigée par les modèles (IDM), l'idée de construire des abstractions qui permettent de manipuler ces systèmes de manière efficace a émergé dans de nombreux travaux [OGT+99],[BBF09a],[ZC06]. Ainsi l'approche désignée sous le terme de "*Model@Runtime (M@R)*" vise à offrir une abstraction qui permet de manipuler les briques logicielles et leurs cycles de vie sous forme d'une représentation modèle, plus simple et moins coûteuse que la manipulation du système réel. Cette couche modèle devient alors une couche réflexive du système réel pouvant être exploitée par divers outils et processus comme les moteurs de raisonnement, ouvrant la voie aux systèmes auto-adaptatifs prenant seuls les décisions d'amélioration de leur état. Une telle couche d'abstraction doit bien expliciter et non masquer les problèmes du domaine d'étude, et ceci est nécessairement vrai dans le monde du distribué en accord avec le constat de Guerraoui et al [Gue99],[GF99] qui parlent de : "mythe de la distribution transparente".

Ainsi le prochain challenge identifié par cette thèse, est d'offrir pour les méthodes de génie logiciel une méthodologie au bon niveau d'abstraction, pour prendre en compte la distribution et l'adaptation distribuée continue d'applications, et ce avec un surcoût limité à la fois d'un point de vue de la puissance de calcul et de la compétence requise par les développeurs de tels systèmes pour ne plus viser seulement de très gros projets fonctionnant sur des super-calculateurs mais également des systèmes d'informations modestes.

0.2 Contribution

En réponse à ce challenge, cette thèse propose une abstraction et un ensemble d'outils fondés sur le principe *Model@Runtime* pour définir, développer et déployer de façon continue les DAS distribués (Distributed Dynamically Adaptive System (DDAS)) sur des nœuds hétérogènes. Pour cela, cette thèse propose trois contributions principales :

- **Une couche générique de modélisation qui couvre de manière cohérente les grandes fonctionnalités d'un DDAS**, à savoir : les nœuds de calculs, les composants métiers, les canaux de communication entre composants, ainsi que leurs capacités de synchronisation. Faisant le lien entre les outils de modélisation et les outils de développement, cette couche d'abstraction rend explicite l'hétérogénéité des implantations pour les différents types de fonctionnalité, depuis des composants déployés sur micro-contrôleurs[FBP+12] jusqu'à des infrastructures de serveurs.
- **L'introduction dans ce modèle d'une entité pour gérer la couche de**

réflexion distribuée et sa sémantique de dissémination. En effet, en distribuant le modèle de réflexion du système distribué sur les différents nœuds qui le composent, il est alors possible de répondre au problème de partage de contexte nécessaire pour l'adaptation distribuée. Pour expliciter les différentes capacités de distribution et dissémination de ce modèle l'approche propose une notion de groupe de nœuds qui encapsule cette sémantique. Inspirée des résultats d'algorithmique distribuée pour les réseaux pairs-à-pairs, cette thèse valide ce mode d'adaptation distribuée[FDP+12b] en exploitant par exemple des approches dérivées du *gossip* [LTB+07],[EGKM04] pour l'adaptation sur réseau pairs-à-pairs. En associant différents groupes à des nœuds du système on assure ainsi différentes propriétés de cohérence comme la consistance éventuelle [TTP+95], [BGL+06].

– **La définition d'opérateurs expressifs de manipulation de modèle d'architectures issus des opérateurs de composition de modèles et des techniques avancées de synthèse de modèles**. En effet, faisant le lien également avec les outils de composition issus de l'IDM, adaptés pour gérer la complexité en largeur, ces travaux proposent différentes méthodes pour manipuler et composer les systèmes d'architecture, depuis des langages dédiés jusqu'à des outils de compositions opportunistes dérivés d'algorithmes génétiques.

L'abstraction proposée dans le cadre de thèse au sein de l'équipe Triskell et décrite dans ce document est nommée Kevoree. Concrétisés sous forme d'un projet Open Source, les travaux autour de cette approche *Model@Runtime* pour les DDAS proposent également une adaptation des outils de modélisation et de conception de systèmes pour une exploitation efficace à l'exécution[FNM+12].

Cette thèse cherche donc principalement à introduire la gestion de la dynamique d'architecture directement dans le cycle de développement, et ainsi passer des approches déclaratives (ADL) vers une programmation orientée architectures. Ceci ouvre la voie à des systèmes réactifs modifiant leur architecture pour fonctionner et accorder leurs fonctionnalités en fonction de leur contexte. Ainsi s'il n'est plus envisageable de concevoir l'architecture d'un système pervasif de façon exhaustive, il sera cependant possible de définir les règles d'adaptation qui vont le construire de manière opportuniste.

Première partie

Contexte, pré-requis et état de l'art

Aucun problème ne peut être résolu sans changer le niveau de conscience qui l'a engendré.
Albert Einstein

Cette partie détaille les motivations de ces travaux de thèse, qui se focalisent sur la recherche d'abstractions pour la construction et l'ingénierie logicielle des systèmes distribués adaptatifs. Comme introduction, le chapitre 1 détaille les problématiques qui ont amené à construire les systèmes large échelle moderne sous formes de systèmes adaptatifs. Ces problématiques qui se retrouvent étendues à cause d'une multitude de nouveaux noeuds de calcul de différentes natures justifient la recherche d'une abstraction et de méthodes pour développer, déployer et maintenir des environnements difficilement maîtrisables. Le chapitre 2 survole quand à lui les différentes technologies utilisées dans ce domaine afin de poser les bases du vocabulaire nécessaire pour présenter l'état de l'art du chapitre 3. Cet état de l'art extrait les critères qui permettent de classifier les différentes approches existantes du domaine avant de synthétiser et conclure sur les perspectives et points de contribution de cette thèse.

Chapitre 1

Introduction

1.1 Complexité grandissante des développements de systèmes d'information

Afin de développer les systèmes d'information aujourd'hui omniprésents dans notre environnement, les outils et méthodes dédiés sont en constante évolution depuis un demi-siècle en réponse à leur complexité grandissante. Cette complexité est influencée par les usages, les volumes d'informations manipulés mais surtout par la capacité à se modeler en fonction des changements d'exigences (plasticité) et la durée de vie toujours plus courte de ces systèmes. A titre d'exemple en 2012, selon une étude de MTVN [1], 38% des applications en lien avec le multimédia mobile auraient une durée de vie inférieure à 3 semaines.

Cette complexité a été le moteur d'une recherche d'abstraction pour développer, manipuler et réutiliser les pièces logicielles qui composent ces systèmes d'information. Une abstraction est un point de vue [FKN+92] cherchant à simplifier une partie d'un problème. Dans un contexte très différent, les peuples ancêtres des polynésiens ont exploité un modèle de navigation maritime basé sur une continuité des parties terrestres et stellaires [BB72]. Si ce modèle est fondamentalement faux d'un point de vue de l'astronomie moderne, ce point de vue a permis des flux migratoires traversant plusieurs milliers de miles nautiques d'océan. La qualité d'une abstraction se ramène donc à sa faculté à expliciter un système pour résoudre un problème donné. Ceci rejoint la citation I d'Einstein dans le sens ou l'abstraction doit diverger de la réalité pour s'approcher au plus près des besoins du problème à résoudre.

C'est dans cette optique et pour répondre à la complexité d'assemblage des systèmes modernes qu'a été imaginée la mouvance de méthodes et d'outils autour de l'ingénierie dirigée par les modèles (IDM). L'IDM appliquée à la construction de logiciel vise donc à fournir des possibilités de points de vue différents et simplifiés d'un système informatique. Ces outils ont d'abord été exploités pour une conception uniquement «contemplative», c'est à dire une modélisation abstraite, simplifiée, d'un système permettant de vérifier et évaluer des propriétés sur ce dernier sans devoir altérer le système

1. http ://goo.gl/5tJJN

réel. Par la suite, l'IDM au travers de sa variante particulière Model Driven Architecture (architecture dirigée par les modèles) (MDA) [2] [KWB03] a été peu à peu rapproché des outils et méthodes de développement principalement via des langages spécifiques à un domaine (Domain-Specific Language (DSL)) accompagné de générateur de code permettant de passer directement du modèle abstrait de simulation à un code machine concret capable de s'exécuter. Ainsi la même couche d'abstraction peut générer du code pour des machines dont les caractéristiques sont très différentes. L'hétérogénéité des plates-formes est donc prise en compte dans ces méthodes via ces multiples générateurs imposant du même coup le cycle de vie unidirectionnel suivant : design du contexte du système abstrait (*"meta-modèle"* ainsi que générateurs), design du cas d'usage (modèle), génération de code exécutable à partir du modèle. Si ce cycle de développement en V répond à la complexité en largeur (i.e. prise en compte d'un grand ensemble de préoccupations), permet-il de faire face à la plasticité grandissante des logiciels ?

1.2 Systèmes éternels et adaptation continue, du cycle en V vers le déploiement continu

Les systèmes informatiques sont devenus en deux décennies l'épine dorsale de multiples organisations humaines. La gravité des impacts économiques et sociaux d'un arrêt ou dysfonctionnement de ceux-ci les ont fait évoluer vers des systèmes «éternels», virtuellement disponibles en permanence, en pratique ceci se traduit par un taux de disponibilité au dessus des 99,9% par an (les fameux trois neuf[VM01] eldorado des centres d'hébergement). Mises en corrélation avec l'évolutivité des usages et la durée de vie très courte des fonctionnalités, ces contraintes imposent de laisser de tels systèmes ouverts à des évolutions non prévisibles lors de la conception initiale. Par exemple, il est aujourd'hui très difficile d'anticiper les prochains services offerts par un aéroport à ses usagers. Il est donc d'autant plus difficile de désigner les connecteurs ou interfaces qui permettront aux nouveaux futurs modules de se connecter et d'exploiter les données de l'aéroport pour fournir ce service. Pourtant dans 5 ans, ce système devra l'intégrer pour les besoins du marché et ce sans interruption ni altération de service.

De nombreux travaux [MSKC04],[MBNJ09b] ont convergé et montré l'intérêt de considérer ces systèmes «critiques» comme des systèmes adaptatifs (DAS). Les DAS intègrent donc dans leurs plates-formes mais également dans leur cycle de développement la notion de mise à jour de leurs fonctionnalités à chaud. Cette catégorie de systèmes impose de repenser l'architecture du logiciel sous-jacent pour prendre en compte cette contrainte afin de ne plus considérer l'exécution de manière monolithique. Pour expliciter la notion de pièce ou fragment logiciel pouvant être mis à jour, les systèmes DAS ont été définis à l'aide de paradigmes de composants ou services. Ce rapprochement a été d'autant plus opportun, car ces systèmes d'information possèdent une forte complexité en largeur, c'est à dire intègrent un grand nombre de fonctionnalités. Le découpage de ces systèmes en composants permet alors non seulement la gestion du cycle de vie mais également comme prévu originellement par cette abstraction dans des ouvrages comme

2. http ://www.omg.org/mda/

celui de Crnkovic *et al* [CL02], de séparer le systèmes en fonctionnalités élémentaires enfin d'en faciliter la maintenance.

Si les DAS ont été réservés à une élite de systèmes critiques il y a quelques années (par exemple : systèmes d'aéroports, bancaires, centraux téléphoniques) ils sont désormais introduits dans des cas d'usage beaucoup plus modestes. Ainsi les box internet, serveurs domotiques[NDBJ08b], serveurs web doivent prendre en compte également ce type d'architecture pour faire des mises à jour continues. Des systèmes non critiques sont maintenant économiquement concernés pour ces besoins. En effet si hier il fallait justifier ce type d'architecture sur un système bancaire dont les pertes s'élèvent à plusieurs millions par minute de *"downtime" (temps d'indisponibilité)*, aujourd'hui avec les développements mobiles ne pas perdre un appel pour cause de mise à jour est un enjeu économique suffisant pour concevoir un système sous forme d'un DAS. Il y a donc un fort besoin d'abstraction pour la construction de ces systèmes adaptatifs puisque l'ouverture à des systèmes non critiques induit des cycles de développement plus courts et plus humbles économiquement.

Les méthodologies de développement ont également connu des modifications profondes pour répondre à la plasticité de ces logiciels. Si le cycle en V était préconisé dans les années 80, à savoir conception et spécification initiale puis génération ou implémentation de code, les méthodes modernes préconisent l'hyper agilité[Sto09]. En effet dans une étude publiée en 2011 [Ver10] sur l'adoption industrielle des méthodes Agiles, *VersionOne* annonce que 80% des sondés déclarent que leur compagnie a adopté un tel processus. Ces nouvelles méthodes cherchent à introduire des cycles plus courts de développement afin de répondre plus rapidement à des évolutions de spécification et surtout à obtenir très tôt des retours utilisateurs. Cette hyper agilité s'accompagne d'une nouvelle approche pour le test et l'intégration des nouveaux développements : l'intégration continue. Dans cette approche un système de tests est remis à jour avec les derniers artefacts de développement en continu, le plus souvent à chaud. Cette méthodologie rapproche d'autant plus ces systèmes non critiques des contraintes spécifiées pour les DAS, étayant la thèse de rapprocher les abstractions des deux mondes.

Le cycle de développement et déploiement se fait donc maintenant de manière continue sur des systèmes critiques ou non. Les approches génératives de l'IDM et surtout de son pendant MDA unidirectionnelle (design vers code) posent donc problème et doivent prendre en compte cette bidirectionnalité afin de répondre au cycle continu. En d'autres termes, le design initial exhaustif préconisé par l'approche en V des années 80 n'est plus envisageable, l'hyper agilité pousse au contraire a étoffer celui-ci à chaque cycle de développement. Le design du système lui-même devient donc continu, les outils doivent suivre ce nouveau cycle de vie. Les générateurs de code devraient alors idéalement fournir leurs opérations inverses pour permettre cycliquement de concevoir le code et coder dans le design. Au delà de cette phrase d'accroche, un des enjeux pour l'IDM est la prise en compte du code patrimonial dans la conception, mais également dans les outils afin d'aller au delà d'une approche descendante conception vers code.

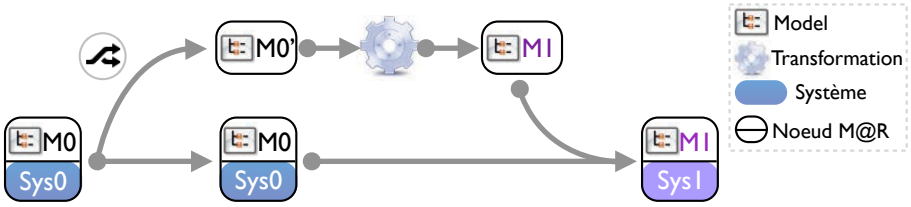
1.3 Mode@Runtime : couche de réflexion de système DAS

En réponse au problème de design cyclique a été proposée l'idée de rapprocher les activités de modélisation et les systèmes concrets via le paradigme nommé *"Model@Runtime"* (M@R). D'abord imaginé encore une fois comme vision contemplative du système à des fins de simulations[OGT+99],[BBF09a],[ZC06], cette approche repose sur une représentation abstraite sous forme de modèles en permanence «à jour» du DAS. Cette couche modèle devient alors une couche d'introspection donnant la capacité d'observer et de raisonner sur l'état du système. Cette couche d'introspection permet une analyse de l'état du système en apportant une navigation aisée dans une représentation schématique (simplifiée) des différents éléments qui le composent. Dans cette tendance du *"Model@Runtime"*, les travaux de Brice Morin[Mor10b] ont promu cette couche modèle non plus uniquement comme couche d'introspection mais également d'intercession permettant de modifier un système au travers de sa représentation réflexive modèle. En accord avec la définition de Bobrow et al[BGW93] l'intercession est la capacité d'un système à modifier son état interne. L'approche M@R de Morin pour les DAS vise justement la construction de système réflexif alliant une capacité d'introspection et d'intercession au travers de la même couche de modélisation. En tant que couche de modélisation abstraite, la couche de réflexion ainsi proposée est désynchronisable, permettant ainsi de faire plusieurs modifications au niveau modèle, et de les tester avant de les appliquer sur le système réel.

De manière schématique cette approche permet d'extraire et modifier le modèle réflexif du système concret (ex : ajout/suppression d'un composant), puis de détecter et déployer toute modification par différence de versions vers la plate-forme (ex : déploiement du composant). Inversement toute modification de la plate-forme extérieure donne lieu à une modification dans la couche de M@R introduisant alors un lien causal bidirectionnel. Le M@R devient donc une couche qui peut être exploitée de manière asynchrone : en d'autres termes, il est possible de modifier celle-ci de manière indépendante de la plate-forme puis la synchroniser par la suite. Ceci permet par exemple d'introduire des étapes de vérification avant déploiement mais surtout de ne pas contraindre les capacités de modification du M@R avec des contraintes de la plate-forme. Par exemple si un composant A dépend d'un autre B, la plate-forme va exiger que A soit déployé avant B pour respecter les contraintes de plates-formes. A l'inverse, l'ajout de B avant A dans le modèle n'est pas soumis aux mêmes contraintes d'ordres si l'application de l'adaptation se fait après les deux ajouts. Les contraintes d'une couche réflexive liée à sa plate-forme d'exécution restreignent alors toutes les approches de génération automatique d'adaptations. A l'inverse en retardant ces contraintes au moment de l'application du modèle réflexif, les approches génératives peuvent ainsi manipuler le modèle dans n'importe quel ordre.

Cette capacité à désynchroniser le modèle réflexif et la plate-forme est illustrée par la figure 1.1 et permet d'exploiter au *runtime* ce modèle comme une cible pour tous les algorithmes de composition qui servent à calculer à partir de divers sources (*"feature model"* (dérivation de fonctionnalités), aspects [MBNJ09b]) un modèle composé de l'architecture du DAS. Le système pouvant décider quand synchroniser le modèle, toutes les

FIGURE 1.1 – Illustration du processus Model@Runtime désynchronisable



opérations avant déploiement *(«offline»)* ne sont pas contraintes par le système concret. Plusieurs travaux [MBJ+09a],[KKR+12],[RBD+09] convergent vers l'utilisation de composant logiciel pour encapsuler les notions de cycles de vie des DAS et leurs opérateurs de composition. Utilisant divers paradigmes de conceptions et compositions pour assembler le système complet, la convergence de ces travaux étaye l'hypothèse à la fois d'une nécessite de diversité de paradigmes de composition mais également de la viabilité d'exploiter ce niveau de granularité (composants) pour gérer les cycles de vie des briques applicatives des DAS.

1.4 Vers des systèmes hybrides distribués, hétérogènes, adaptatifs

Ces cinq dernières années la plasticité des systèmes a connu une forte accélération due à des changements profonds des usages eux-mêmes influencés par plusieurs facteurs. D'une part le nombre de types de terminaux qui permettent d'interagir avec ces systèmes d'information nouvelle génération a fortement augmenté notamment pour répondre aux besoins de mobilité : selon une étude de Gartner [Gar10], les ventes de Smartphones augmentent de 96% entre 2009 et 2010 pour atteindre 80 millions d'unités vendues par trimestre. Dans cette continuité, un ensemble de matériels regroupés sous l'appellation Internet des Objets est venu se greffer à ces systèmes complexes. Directement issu de l'électronique embarquée ces «capteurs/actionneurs» basse consommation connectent les systèmes informatiques à leur contexte physique. Les systèmes dédiés à l'hébergement des systèmes d'information eux-mêmes (DataCenters) connaissent également une phase de transition. L'émergence des techniques de mutualisation nommées sous le terme Cloud computing implique de repenser le développement de logiciel monolithique en système distribué. En effet afin de pouvoir profiter des services d'élasticité offerts par ces infrastructures, les logiciels doivent repenser les accès aux données, mais également la façon de séparer en sous-systèmes autonomes l'application en intégrant les contraintes imposées par le cycle de vie de ces sous-systèmes.

À ces nouveaux usages viennent s'ajouter les nouveaux modèles économiques. En effet si la loi de Moore (prédiction de l'évolution de la puissance de calcul) tend à ne plus s'appliquer, la durée de vie moyenne d'un matériel informatique tend à se réduire, rendant du même coup aussi court le temps d'obsolescence du logiciel tournant dessus (environ tous les ans pour les systèmes Android). Ce phénomène s'accentue d'autant

pour tous les matériels basse consommation qui composent l'Internet Of Things (IoT). Ainsi comme il est expliqué par Ko and al [PCBD10], chaque cas d'application peut remettre en question la balance entre performance et consommation et donc le choix du type de plate-forme.

La distribution, à savoir la répartition des constituants d'un système d'information sur les différents noeuds qui le composent est donc devenue inévitable et doit être prise en compte par les approches de génie logiciel. Là encore si ces besoins étaient il y a peu réservés à des élites de super calculateurs ou grilles de calcul il est aujourd'hui nécessaire pour des simples applications de gestion de prendre en compte ce besoin rendant là encore le besoin d'abstraction nécessaire pour la construction de ces systèmes.

La distribution des plates-formes induit de manière sous-jacente la nécessité de prendre en compte l'hétérogénéité de celles-ci. En effet Morin and al [MBNJ09b] avaient identifié le problème de composition lorsque l'on prend en charge tous les points de variation d'un système dynamique mono-plateforme, ce problème est aggravé par la multiplicité et donc les nouveaux points de variation apportés par les différents noeuds d'exécution.

1.5 Challenges et points de contribution

Les systèmes d'information (SI) évoluent et leurs modèles d'architecture convergent vers ceux des systèmes distribués adaptatifs. Ces systèmes deviennent de fait **distribués** et **hétérogènes** car ils sont déployés sur de nombreux terminaux de types différents (des grilles du Cloud jusqu'aux Smartphones). Ils deviennent également adaptatifs continus car doivent prendre en compte de manière continue pendant le fonctionnement des nouvelles fonctionnalités.

L'adaptation d'un système est définie par Georgas *et al*[GT04] comme une réaction à un stimuli en prenant en compte le contexte d'exécution pour calculer le prochain état du DAS. Ce contexte d'exécution qui peut être rapproché d'une couche de réflexion impose donc d'avoir un état : une représentation globale ou partielle du système distribué qui permet alors de calculer le prochain état du DAS. La maintenance de la cohérence de cet état dans un environnement distribué est un problème difficile et particulièrement dans des environnements mobiles où les connections sporadiques (intermittentes) rendent difficile la garantie de mise à jour de ces derniers. De manière opposée les noeuds d'un DataCenters doivent eux garantir des modifications coordonnées dans le même temps logique. Enfin dans un dernier exemple, dans le cas des réseaux de capteurs la synchronisation directe de la couche de réflexion a un impact très négatif sur la durée de vie de ces périphériques autonomes fonctionnant sur batterie. Ces trois cas de figure peuvent bénéficier d'une approche asynchrone de reconfiguration permettant à la fois de limiter les communications mais également d'adopter différentes stratégies de dissémination du modèle d'architecture.

Les travaux présentés dans cette thèse proposent une méthode de construction pour définir et gérer des modèles d'architecture de DAS distribués (DDAS, Distributed Dynamically Adaptive System). S'inscrivant dans la continuité des travaux sur le M@R

de Brice Morin ces travaux proposent une couche d'abstraction modèle comme couche de réflexion et d'intercession, afin de permettre la gestion des DDAS suivant toutes les approches de compositions d'IDM. Organisée autour de trois contributions principales, cette thèse propose :

- Une couche générique de modélisation qui permet la modélisation des grandes fonctionnalités d'architecture d'un DDAS hétérogène.
- L'introduction dans ce modèle d'une entité de première classe pour gérer la couche de réflexion distribuée et sa sémantique de dissémination.
- La définition d'opérateurs expressifs de manipulation de modèle d'architectures issus des opérateurs de composition de modèles.

Ces propositions sont synthétisées dans une abstraction nommée Kevoree, qui combine une approche Model@Runtime et un modèle de composant pour la modélisation complète des éléments d'un DDAS.

Le modèle proposé permet de représenter et déployer les principales notions des DDAS : les noeuds de calcul (*nodes*), les composants, les canaux de communication (*channels*), ainsi que leurs points de synchronisation (*groups*). L'approche proposée permet alors par compositions de ces différentes notions d'offrir une approche générique pour la construction des DDAS. Les *Nodes*, conteneurs de composants, encapsulent la sémantique d'adaptation (comment déployer un composant, le mettre à jour, etc...) tandis que les composants encapsulent eux les fonctionnalités métiers. Les canaux de communication permettent de modéliser et d'expliciter les différents canaux de communication offrant différentes garanties de service. Enfin les groupes de communication permettent de définir les sémantiques et la portée de synchronisation du modèle du système. En associant ces groupes à des noeuds du système on assure ainsi la cohérence de leur configuration et modèle commun. Durant cette thèse, sont explorées différentes stratégies et algorithmes pour la synchronisation des groupes. Notamment, les approches Gossip[LTB+07],[EGKM04] dérivées des algorithmes utilisés pour la propagation des informations dans les réseaux sociaux ont été validées pour des usages sur des réseaux pairs-à-pairs.

En offrant une modélisation des différents types de noeuds de calcul mais également de leurs différents types de synchronisation et communication ce projet offre l'abstraction nécessaire pour traiter de manière homogène le problème d'**hétérogénéité** et d'**adaptation distribuée** des DDAS. Cette approche propose d'introduire la gestion de la dynamicité directement dans le cycle de développement mais également dans la gestion des modèles d'architectures passant ainsi des approches déclaratives (ADL) vers une programmation orientée architectures. Ce type de programmation ouvre la voie vers des systèmes réactifs exploitant et modifiant leur architecture pour fonctionner et accorder leurs fonctionnalités en fonction de leur contexte environnant. S'il n'est plus envisageable de concevoir un système distribué de façon exhaustive, il est cependant possible de définir les règles d'adaptation qui vont le construire de manière opportuniste.

1.6 Organisation du document

Ce document est composé de quatre parties principales. La première place le contexte de la thèse et dresse un état de l'art des langages de description d'architectures, des modèles de composant offrant des capacités de réflexion, des modèles d'agent qui permettent la distribution des traitements et des algorithmes qui permettent de distribuer et coordonner des données sur des topologies de réseaux complexes et dynamiques. La seconde partie présente les concepts de l'abstraction proposée nommée Kevoree. La troisième partie propose trois niveaux de validation de ce modèle pour démontrer son utilisabilité dans un contexte de déploiement constitué de nœuds hétérogènes, dans un contexte de déploiement sur des topologies de réseaux complexes et dynamiques et pour discuter son utilisabilité par une communauté de développeurs. Enfin, une dernière partie regroupe les conclusions, contributions et perspectives de ce travail.

Partie 1 : « État de l'art »

Le **chapitre 1** « Introduction » présente les défis de l'ingénierie et l'architecture logicielle qui motivent l'ensemble des travaux de cette thèse.

Le **chapitre 2** « Contexte » présente les grandes approches communément utilisées pour répondre aux développements pour des plates-formes hétérogènes et distribuées, un cas d'étude y est également présenté permettant de mettre en lumière les contributions.

Le chapitre 3 « État de l'art » dresse l'état de l'art des projets permettant la construction de systèmes réflexifs ou distribués. Ce chapitre se conclue sur les challenges identifiés qui motivent les contributions de cette thèse.

Partie 2 : « Thèse et Application »

Le chapitre 4 « Concepts généraux de la contribution » présente les concepts généraux qui introduisent la contribution de cette thèse.

Le chapitre 5 « Modèle de composant étendu » présente les éléments de l'abstraction Kevoree issus des modèles composants.

Le chapitre 6 « Model@Runtime distribué » présente les éléments de l'abstraction Kevoree permettant la réalisation et la coordination d'un processus Model@Runtime dans un environnement distribué.

Partie 3 : « Validation »

Le chapitre 7 « Axes d'évaluation » introduit et justifie les axes sur lesquels l'abstraction a été évaluée.

Le chapitre 8 « Evaluation quantitative aux limites : gestion des *Cyber Physical Systems* » présente l'axe d'évaluation sur environnements hétérogènes en prenant l'exemple des CPS.

Le chapitre 9 « Évaluation quantitative aux limites : adaptation de DDAS en environnement mobile » présente l'axe d'évaluation sur environnement distribué sur la base d'une propagation de type *gossip*.

Le chapitre 10 « Ecosystème Kevoree et validation par les projets liés » présente l'ensemble des éléments permettant de valider la maturité et la généricité du projet Kevoree au travers de collaborations mais également d'une évaluation sur un panel de développeurs.

Le chapitre 11 « Conclusion de validation » conclue la partie validation en tirant les conclusions des différentes données expérimentales récoltées.

Partie 4 : « Conclusion et perspectives »

Le chapitre 12 « Conclusion » résume et conclue sur les résultats issus de cette thèse.

Le chapitre 13 « Perspectives » énonce les différents axes de perspectives de recherche envisagés à la suite de ces travaux, notamment pour répondre à la synthèse d'architecture continue ou encore pour la gestion des architectures de type *cloud computing*.

1.7 Liste des publications liées à cette thèse

Revue internationale

– Jean-Marc Jezequel, Benoit Combemale, Olivier Barais, Martin Monperrus, Francois Fouquet – Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench – Software and Systems Modeling (in major revision)

Communications internationales

(Avec édition d'actes et comité de sélection)

– F. Fouquet, O. Barais, N. Plouzeau, J-M. Jézéquel, B. Morin and F. Fleurey. – A Dynamic Component Model for Cyber Physical Systems. – In CBSE'12 : 15th International ACM SIGSOFT Symposium on Component Based Software Engineering. Bertinoro, Italy, June 2012.
– Fouquet F., Daubert E., Plouzeau N., Barais O., Bourcier J., Jézéquel J.-M. – Dissemination of reconfiguration policies on mesh networks – In DAIS'12 : 7th International Federated Conference on Distributed Computing Techniques, 2012, Stockholm
– F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau and J-M. Jézéquel. – An Eclipse Modeling Framework Alternative to Meet the Models@Runtime Requirements. – In MODELS'12 (Application Track) : ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, Innsbruck, Austria, October 2012.
– Viet-Hoa Nguyen, Noël Plouzeau, François Fouquet and Olivier Barais. – A Process for Continuous Validation of Self-Adapting Component Based Systems. – In MRT 2012 MODELS'12 workshop) : ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, Innsbruck, Austria, October 2012.
– E. Daubert, F. Fouquet, O. Barais, G. Nain, G. Sunyé, J-M. Jézéquel, J-L. Pazat and B. Morin. – A models@runtime framework for designing and managing Service-Based

Applications. – In ICSE Workshop on European Sofware Services and Systems Research - Research and Challenges (S-Cube). Zurich,Switzerland, May 2012.
- G. Nain, F. Fouquet, B. Morin, O. Barais and J-M. Jézéquel. – Integrating IoT and IoS with a Component-Based approach. – In SEAA'10 : 36th Euromicro Conference on Software Engineering and Advanced Applications, Lille, France, September 2010.
- F. Fouquet, O. Barais, J-M. Jézéquel – Building a Kermeta Compiler using Scala : an Experience Report. – In Workshop Scala Days 2010. Lauzanne, Switzerland 2010

Communications nationales

(Avec édition d'actes et comité de sélection)
- François Fouquet, Erwan Daubert, Noël Plouzeau, Johann Bourcier, Jean-Émile Dartois, Olivier Barais, Arnaud Blouin – Kevoree : une approche models@runtime pour les systèmes ubiquitaires – UbiMob'12, Anglet, France, 2012

En cours de soumission

- Francois Fouquet, Johann Bourcier, Benoit Baudry and Noel Plouzeau – An Efficient Multi-Objective Optimization Genetic Algorithm for Pervasive Systems – PerCom 2013
- Francois Fouquet, Erwan Daubert, Olivier Barais, Brice Morin, Franck Fleurey, Noel Plouzeau, Jean-Marc Jezequel – Seamless Adaptation of Complex Heterogeneous Systems : from Cloud Servers to Cyber Physical – SciCo special issues 2013
- Gregory Nain, Francois Fouquet – Tutorial on Kevoree – ICSE 2013
- Jean-Marc Jezequel, Benoit Combemale, Olivier Barais, Martin Monperrus, Francois Fouquet – Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench – SOSYM
- Gerson Sunye, Eduardo Cunha de Almeida, Erwan Daubert, Francois Fouquet, Olivier Barais – Macaw : an Architecture for Testing Large-Scale Dynamic Distributed Systems – Computing Journal 2013

Chapitre 2

Contexte

La contribution de cette thèse vise à résoudre deux problèmes majeurs pour le développement logiciel de systèmes adaptatifs : l'hétérogénéité des plates-formes d'exécution et l'utilisation des ressources de calcul distribuées. Ces besoins sont particulièrement important dans les systèmes d'information de terrains dont une illustration est donnée avec le projet DAUM détaillé en section 2.1. Le reste du chapitre détaille les définitions des grandes approches et modèles de développement communément exploités pour répondre aux deux problèmes identifiés : développement hétérogène 2.2 et distribué 2.3. Ces définitions seront par la suite exploitées dans le chapitre 3 dressant l'état de l'art du domaine.

2.1 Cas d'étude : projet DAUM

Depuis 2009, des réflexions ont été engagées avec les pompiers du Service Départemental d'Incendie et de Secours d'Ile-et-Vilaine (SDIS35) pour l'élaboration d'un système tactique de terrain informatisé pour les opérations d'urgence. Sur bien des aspects ce type de système rentre dans la catégorie de systèmes visée par la contribution de cette thèse. Le projet Dynamic Adaptation Using Models (DAUM) qui résulte de ces réflexions a joué un double rôle. D'un coté c'est un cas d'étude qui permet d'extraire et d'illustrer les besoins d'un système adaptatif distribué, de l'autre c'est une étude de validation puisque le prototype du projet a été réalisé avec les travaux de cette thèse.

2.1.1 Un besoin d'information sur le terrain

Sur le terrain et lors d'une intervention, les pompiers ont besoin d'échanger des informations en temps réel sur la situation des moyens, des personnels engagés et sur les actions en cours et à venir. Les données tactiques des actions et moyens sont synthétisées via une notation graphique nommée Tactical Reasoning Method (TRM). Actuellement, ces données collectées sont échangées via des canaux de communication radio et reproduits sur des tableaux blancs dans des centres de commandement pour illustrer la situation courante et tracer l'historique des positions des moyens mis en oeuvre. Sur le terrain les groupes de personnels sont organisés sous la forme d'une arborescence

hiérarchique. Un chef est donc désigné, ayant sous ses ordres un groupe d'hommes et des sous-groupes sur lesquels il a la visibilité des informations tactiques.

L'adaptation est inhérente au fonctionnement des pompiers, en effet les missions montent en puissance en fonction des besoins, ajoutant ainsi sur place du personnel, et des véhicules. Ces changements impactent du même coup l'organisation hiérarchique, géographique et donc la topologie des échanges radio associés.

2.1.2 Des noeuds de calcul et un réseau maillé de terrain

La solution envisagée par le projet DAUM consiste à informatiser l'ensemble de la chaîne de collecte et diffusion des informations tactiques mais également de les enrichir avec des informations biométriques sur les personnels. Pour cela, chaque responsable d'un secteur et chaque officier impliqué dans la chaîne de commandement doit disposer sur le terrain d'une tablette tactile ou d'un tableau tactile pour présenter et éditer la situation tactique. Ces tablettes doivent s'organiser en topologie réseaux pairs-à-pairs pour déployer sur le terrain un réseau maillé permettant de résister à la perte de noeud. En effet tout matériel utilisé en intervention a de grande chance d'être endommagé ou d'être simplement hors connexion aboutissant à une topologie réseau très dynamique.

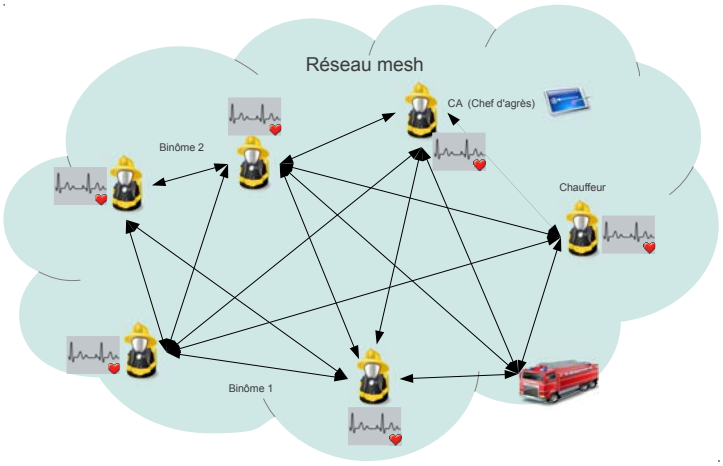
A cela s'ajoute un équipement embarqué dans les vêtements de sécurité des personnels. Reliés à des capteurs biomédicaux (rythme cardiaque, analyse de température corporelle, de stress), de positionnement géographique (GPS, triangulation réseau) et d'analyse des températures environnantes (température du sol et plafond) ces dispositifs sauvegardent l'ensemble des informations récoltées par un porteur. Ils peuvent également s'adapter suivant les différentes interventions (risque chimique, feu, etc) et avertir le porteur d'évènements urgents.

Enfin des ordinateurs de soutien permettent de stocker et redistribuer les données récoltées des informations tactiques et des données des capteurs de terrain. Ces noeuds sont embarqués dans les véhicules et participent aux échanges de données dans un réseau maillé de terrain entre les noeuds de capteurs, les tablettes et les ordinateurs de soutien. La sauvegarde et la traçabilité historique des données est un besoin légal vis à vis de la législation française, ces noeuds sont synchronisés pour la même raison avec les centres de commandement régionaux (CODIS). La figure 2.1 illustre le réseau de terrain global envisagé pour les interventions.

2.1.3 Challenge de modélisation de la solution

Ce type d'architecture met particulièrement en lumière les besoins de gestion d'adaptation dans les plates-formes hétérogènes. En effet les tablettes de commandement exploitaient des architectures à basse/moyenne consommation sur des processeurs de type ARM, tandis que les noeuds de soutien, alimentés par plus d'énergie peuvent être plus puissants. A l'opposé, les noeuds capteurs dans les vêtements sont des architectures à très basse consommation de type micro-contrôleur. La distribution du logiciel sur des noeuds hétérogènes est donc inhérente au cas d'usage qui doit faire collaborer ces noeuds dans un réseau commun. L'adaptation est inhérente au métier même des pompiers, en

FIGURE 2.1 – Illustration d'organisation de terrain des différents noeuds



effet tous les noeuds doivent faire face à des reconfigurations dynamiques. En effet, les noeuds tablettes doivent changer leur mode de communication suivant le rôle du porteur tandis que les noeuds capteurs changent le type et la fréquence de mesure suivant le type de sinistre ainsi que le type de communication (radio longue ou courte portée) de façon opportuniste. Tous les noeuds doivent également faire face à des changements de topologie qu'ils soient dus à l'arrivée de renfort ou à la perte d'un noeud en défaillance, ce qui dans ce contexte hostile est plus qu'envisageable.

Pour conclure cette architecture nécessite réellement une conception qui lui permet de se surveiller afin de parer aux aléas de fonctionnement et d'assurer en cas d'urgence un service minimum, ce qui est typiquement le but d'un système DDAS tel qu'envisagé dans cette thèse. Ainsi l'objectif de cette thèse est de fournir une abstraction support pour le développement de logiciels répondant à ce type de problèmes, pour permettre par exemple de déplacer des composants logiciels d'une tablette à une autre, ou encore afin de les faire collaborer pour faire de l'agrégation des informations de terrain.

2.2 Développement pour plate-forme hétérogène

Les problèmes de développement pour des plates-formes différentes est un problème récurrent qui tire ses racines depuis la standardisation des premières architectures matérielles. En effet depuis l'opposition entre les architectures PowerPC et x86 les subtilités d'exécution telles que les environnements de mémoire *big* et *little endian* poussent les langages de développement à fournir des solutions pour s'en abstraire.

Les mécanismes permettant de s'abstraire des spécificités matérielles sont multiples et possèdent chacun leurs atouts. Les approches par langages interprétés ou par machine virtuelle, interprètent tardivement une représentation abstraite commune, tandis que les approches par frameworks et par méthodes génératives issues du monde MDA visent à encadrer les primitives données aux développeurs pour éviter d'exploiter du code

spécifique.

Cette section passe en revue ces mécanismes couramment utilisés afin de pouvoir expliquer les choix de solution pour gérer l'hétérogénéité dans l'étude de l'état de l'art mais également dans l'abstraction et la méthodologie proposée.

2.2.1 Abstraction et framework

Une première approche afin de lisser et masquer les différences d'exécution des plates-formes est d'exploiter un framework qui définit un sous-ensemble commun de ce qui est manipulable (API) et fournit ensuite une implantation ou interprétation de ce sous-ensemble pour un certain nombre de plates-formes. Cette approche a deux avantages, elle est extensible : par nature on peut fournir de nouvelles implantations pour de nouvelles plates-formes. Mais surtout elle découple le code métier des implantations en forçant le développement au travers de l'API, ceci permettant après coup de choisir de déployer avec telle ou telle plate-forme. La limitation qui en découle est que l'API borne les possibilitées des utilisateurs qui ne peuvent alors plus exploiter les spécificités des plates-formes par exemple pour faire des optimisations. Néanmoins cette approche est largement exploitée depuis les premiers langages assembleurs (par exemple le framework d'accès aux API de l'OS Windows [1]) jusqu'aux frameworks actuels, pour du calcul parallèle à haute performance (ex : OpenCL [2]) ou des environnements de développement 3D (ex : OpenGL http ://www.opengl.org/).

2.2.2 Langage interprété

Les langages dynamiquement interprétés sont également exploités pour des exécutions inter plates-formes. En effet cette catégorie de langage est interprétée entièrement lors de son exécution, la sémantique d'exécution ainsi que le mapping avec les instructions bas niveau sont donc réalisés par l'interpréteur pour lequel on fournit une implantation par plate-forme. Souvent associé à des langages faiblement typés, l'exemple le plus connu de ce mécanisme est certainement JavaScript [GME07] mais on peut également y classifier Python, Ruby ou Lisp.

2.2.3 Machine virtuel et Just-In-Time compiler

Les machines virtuelles (VM) sont en un sens une extension de l'approche par framework visant à abstraire l'ensemble des instructions bas niveaux d'un matériel (ex : allocation de mémoire, etc ...). A l'inverse d'exécuter du code natif pour le processeur, les machines virtuelles définissent un jeu d'instructions intermédiaires qu'elles sont capables d'exécuter à l'aide d'instructions natives. Cette couche intermédiaire peut alors être exploitée comme sortie des compilateurs de codes sources. Pour chaque plate-forme visée, il est nécessaire de fournir une implantation de VM qui peut alors effectuer des optimisations avec le matériel local au moment de l'exécution (Just-In-Time compiler),

1. http ://www.acm.uiuc.edu/sigwin/old/workshops/winasmtut.pdf
2. http ://www.khronos.org/opencl

cette technique visant à réduire l'impact sur les performances de l'interprétation du code intermédiaire. Cette technique exploitée par exemple pour le langage Java a permis à la société Sun de concrétiser son slogan : “write once, run everywhere”.

2.2.4 Approche générative et IDM

2.2.4.1 Générateur au design

A l'inverse des langages interprétés tardivement à l'exécution, les approches génératives visent à exploiter toutes les informations des abstractions de développement (langage ou modèle) afin de générer dès le design, du code natif adapté à chaque plate-forme d'exécution visée. De tels développements se font alors en deux étapes : d'une part la définition de langage spécifique au cas d'usage ainsi que ses générateurs, puis le développement proprement dit du logiciel à l'aide des abstractions fournies. En mode génératif, l'application doit être intégralement composée avant la génération, qui peut alors effectuer toutes les vérifications et optimisations en profitant du “monde clos”. Point central de cette approche, le langage spécifique est également le point critique, en effet toute évolution de ce langage impacte immédiatement tous les développements précédemment effectués ainsi que les générateurs de code. Le développement de ces DSL est donc critique et outillé par des approches telles que l'IDM.

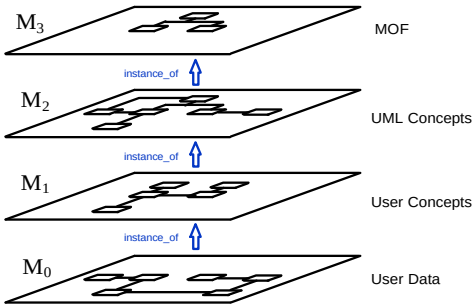
2.2.4.2 Principes et généralités de l'IDM

L'idée fondamentale de l'IDM est la recherche d'une abstraction la plus *"naturelle"* à manipuler pour un cas d'usage donné. Dans cette optique l'abstraction prend la forme d'un modèle qui synthétise alors l'essentiel des informations pour ce cas d'usage, rendant alors cette représentation modèle plus simple et plus adaptée que le système réel en cours d'étude. La distance entre le système réel et sa couche de modélisation devenant potentiellement trop grande et donc trop complexe justifie l'usage de modèle intermédaire. C'est dans cette optique qu'a été introduit le Meta Object Facility (MOF)[OMG11] afin de standardiser une architecture en couche générique. Chaque couche exploite et dérive ainsi les concepts des couches supérieures, les modèles sont alors instances de la couche supérieure qu'on nomme alors méta-modèle, cette organisation est illustrée par la figure 2.2. Cette pyramide de modèles s'étend du niveau le plus élevé décrit à l'aide de lui-même en MOF M3, jusqu'au niveau le plus bas M0 (modèle en cours d'étude), elle est détaillée avec plus de précisions par Atkinson and al [AK03a]. Point essentiel de l'IDM cette approche en couches permet de définir des outils au niveau méta (ex : générateur de code, interpréteur, compilateur) qui seront alors valides pour tout modèle instance (utilisant les concepts de) ce méta-modèle. Cette pyramide optimise donc la réutilisation des concepts tout en permettant de les raffiner à chaque couche.

2.2.4.3 IDM pour les approches génératives

L'IDM est un outil d'aide pour les méthodes génératives et notamment pour le développement de langages spécifiques. En effet en raffinant les concepts existant du MOF

FIGURE 2.2 – Hiérarchie des couches de modélisations (extrait de [AK03b])



dans un méta-modèle on définit les concepts du nouveau langage. Il est alors possible de réutiliser tous les frameworks définis au niveau du MOF permettant la génération d'outils de manipulation textuel [3] [4] ou graphique [5] des modèles instances. Outre la réutilisation d'outils de base pour la construction de langage, un ensemble d'opérateurs de composition et migration de modèles permettent d'améliorer la maintenance de ces nouveaux langages

L'IDM place donc le modèle comme pièce centrale de la conception des approches génératives à la fois pour la génération du code proprement dit à l'aide de transformation de modèles mais également pour la construction des étapes de validation (modèle checking). Ce type d'approche est largement utilisé notamment pour les logiciels critiques tels que les contrôles commandes avioniques.

2.2.4.4 Approche mixte API et génératif

Les Langage de Définition d'Interfaces (IDL) sont une réponse au besoin de communication entre applications hétérogènes. Ces langages définissent à la fois une API et un langage d'assemblage des points de communication des applications. Ces IDL abstraits sont alors exploités par des méthodes génératives afin de générer le bus de communication entre deux plates-formes. Ce type d'approche a été exploité par le projet *CORBA* standardisé par l'OMG [6].

2.3 Développement pour environnement distribué

Depuis l'avènement des réseaux informatiques après les années 60, de nombreux concepts ont été développés afin de permettre les échanges de données entre les différentes machines de ces réseaux. La complexité de développement induite par ces communications réseaux a poussé au développement d'abstractions permettant d'aligner ces

3. http ://www.emftext.org
4. http ://www.eclipse.org/Xtext
5. http ://www.eclipse.org/modeling/gmf
6. http ://www.omg.org/

appels réseaux avec les langages de développement. Ces abstractions ont largement influencé les abstractions logicielles construites au-dessus, et notamment les approches à composants et agents. Cette section dédiée aux abstractions pour le développement distribué, rappelle donc les problèmes d'alignement d'échanges réseaux avec les langages impératifs ainsi que les solutions qui ont suivi pour masquer cette complexité, telles que les modèles à évènements ou encore les RPC 2.3.4. Les systèmes distribués ne se résument pas à la communication entre deux noeuds, des abstractions ont donc été développées pour manipuler les échanges entre systèmes, et ainsi capitaliser sur la notion de patron d'architecture, ceci est détaillé dans la sous-section 2.3.7. Finalement cette section se termine avec une description des approches algorithmiques dédiées à la gestion d'échanges sur des topologies réseaux complexes ou pour assurer la synchronisation de plusieurs machines. La sous-section 2.3.8 rappelle les approches de synchronisation par consensus tandis que la sous-section 2.3.9 détaille les approches épidémiques.

2.3.1 Processus et échange synchrone et asynchrone

Les styles d'échanges synchrone et asynchrone et leurs impacts sur les langages de programmation, expliquent en partie la complexité des styles d'architecture répartis. En effet fondamentalement les supports réseaux physiques échangent les données binaires en mode asynchrone, c'est-à-dire sans synchronisation d'horloge commune, par opposition aux échanges synchrones. Les deux extrémités du réseau peuvent alors avoir des cadences différentes. Paradoxalement, à quelques exceptions près, les langages de programmation et notamment les types impératifs sont foncièrement synchrones. À la différence du domaine du réseau, le terme synchrone désigne ici non plus une horloge électronique commune mais une synchronisation des processus logiciels pendant leurs échanges. Cette différence d'approche entre les processus logiciels et leurs supports réseaux s'explique par plusieurs raisons.

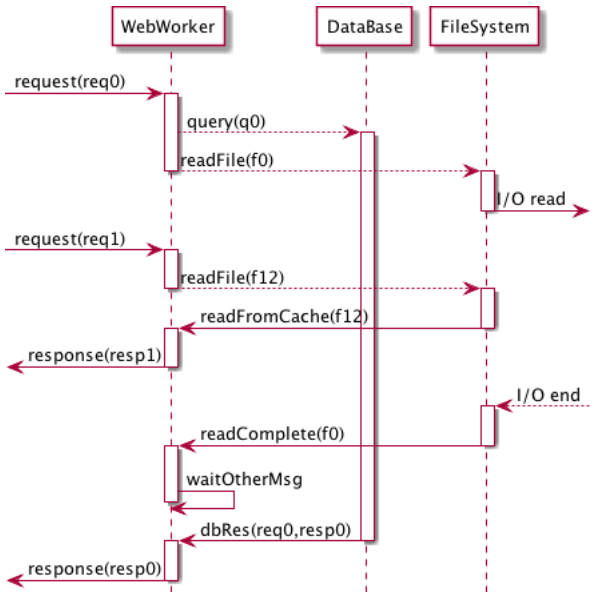
D'abord, historiquement les architectures matérielles ont été séquentielles, et ses processus exécutent donc séquentiellement le code machine. Ce style de programmation était donc plus proche du matériel, ceci est largement remis en question avec les architectures multi-core modernes et explique la remise en question de ce modèle. De plus, le modèle synchrone est considéré comme plus facile à maîtriser que son pendant asynchrone [FMG02] (car permet une représentation mentale du fil d'exécution), à formaliser et son exécution plus prévisible pour les développeurs. Finalement le code synchrone est résolument proche des processus, là où le code asynchrone est lui résolument proche des flux d'échange de données. Le *framework* AJAX [G+05], dédié à l'échange de données entre un navigateur Web et un serveur illustre ce choix de l'asynchrone pour les échanges. La complexité d'usage de processus asynchrones dépend donc essentiellement du type d'échanges du système.

2.3.2 Modèle de programmation évènementiel

Un modèle de programmation évènementiel permet de composer différents traitements par le biais d'envoi et d'écoute de messages asynchrones. L'exécution ne s'écrit

donc pas de façon séquentielle mais plutôt comme une succession de traitements réactifs qui doivent se composer derrière une ou plusieurs boucles d'écoute (*handler* d'évènements). Les traitements s'abonnent à une boucle d'évènements pour être déclenché lors d'un envoi. Calqué sur le modèle des échanges réseaux mais aussi sur les premières interruptions matérielles, la programmation par évènements peut être exploitée de façon plus généraliste comme c'est le cas dans le récent projet NodeJS [7] qui par exemple généralise l'accès à un système de fichiers derrière un modèle évènementiel. La figure 2.3 illustre par un diagramme de séquence le fil d'exécution d'un serveur Web dirigé par des évènements tel que NodeJS. De manière analogue au chapitre précédent, la complexité d'un tel modèle de développement réside dans le caractère asynchrone des échanges, illustré par exemple par le fil d'exécution du *WebWorker*. En effet ce traitement doit attendre la totalité des messages pour renvoyer une réponse sans pouvoir faire de supposition sur l'ordre d'arrivée, ce qui mène à des cas de multiplexage de traitement (par exemple *req1* est effectué pendant *req0* dans la figure) qui peuvent être complexes à gérer de manière analogue au multiplexage des trames réseaux.

FIGURE 2.3 – Diagramme de séquence d'un serveur Web en mode évènementiel



2.3.3 Socket et port réseau

Les *Berkeley sockets* [SFR04] ont été définies pour permettre la communication entre processus sur une même machine ou entre machines distantes. Introduites dans les années 80, elles sont implémentées dans la quasi-totalité des langages et forment la base de la communication actuelle. Les *sockets* permettent d'implémenter un modèle

7. http ://nodejs.org/

client-serveur brut : en d'autres termes, elles permettent d'échanger des données binaires non structurées et sous forme synchrone, leurs API est inspirée des accès fichiers locaux.

Complexes à utiliser dans un langage procédural car orientées flux, les *sockets* peuvent cependant être masquées derrière un *framework* qui simule des échanges évènementiels asynchrones au dessus d'elles comme le projet NodeJS ou le protocole WebSocket [8] [9]. Ces évènements masquent aux développeurs la complexité de traitement du flux bi-directionnel de données entre client et serveur et la gestion inhérente des zone tampons nécessaires (*buffering*). Enfin, le traitement de ces évènements se fait derrière une boucle unique assurant le traitement séquentielle, fournissant ainsi une gestion explicite des accès concurrents sur le serveur.

2.3.4 *Remote Procedure Call* et Service

Le modèle de communication par *socket*, bien que fournissant une couche de base pour les échanges réseaux n'est pas aligné avec le caractère synchrone des langages impératifs. Les Remote Procedure Call (RPC) sont un alignement des appels de procédures des langages à objet sur des appels réseaux bas niveaux tels que les *sockets*. Cette sur-couche permet alors de façon transparente dans un langage impératif d'appeler une méthode locale ou distante sur un serveur. Les RPC ont été la base des architectures à services Service Oriented Application / Architecture (SOA). En effet un service est défini comme un ensemble de méthodes fournissant une fonctionnalité déployée sur un ou plusieurs serveurs. Les RPC sont fondamentalement synchrones et donc s'intègrent aisément dans un langage à objet procédural. Cependant les architectures SOA masquent la complexité des appels réseaux sans offrir d'abstraction pour la gestion des erreurs. Des langages d'orchestration comme BPEL [10] dédiés au traitement synchrone des aléas de communication sont donc apparus pour construire des processus dédiés au contrôle des échanges entre services.

2.3.5 Bus de messages

A l'inverse des RPC les Message-Oriented Middleware (MOM) s'inscrivent dans une volonté de construire tous les échanges de données autour du paradigme non plus de service mais de message. Un message est ici un conteneur de données et l'ensemble des échanges du système se traduit par les envois et réceptions de ces structures. Fondamentalement asynchrones les MOM permettent de fédérer un ensemble de processus autour d'une zone d'échange commune qui offre alors des points d'inscription. Ce type de middleware offre notamment deux types de sémantique d'échange. Le mode "queue" permet d'implémenter le mécanisme de file d'attente concurrent, un producteur dépose donc un message et un des processus consommateurs reçoit et consomme le message. A l'inverse le "Publish and Subscribe" permet de s'inscrire à un point d'intérêt et redistribue à tous les consommateurs tous les messages émis sur ce point. Ce type d'architecture orientée

8. http ://www.websocket.org/
9. http ://dev.w3.org/html5/websockets/
10. http ://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

donnée et message a été exploité notamment pour l'intégration de système d'information d'entreprise (EAI/ESB) mais également directement inclus dans des langages comme Erlang [11] [AVWW96].

2.3.6 Gestion de processus concurrents locaux et distants

Avec l'avènement des systèmes multitâches des année 80, puis des processeurs multi-core actuels, la problématique d'expression de programme concurrent est maintenant inévitable [Sut05]. En effet si virtuellement les systèmes multitâches faisaient tourner en parallèle plusieurs processus, les processeurs multi-core actuels peuvent eux réellement exécuter des instructions en parallèle. Plusieurs paradigmes permettent d'exprimer la notion de programme parallèle avec différents outils, chacun amenant un degré différent de coût en terme de performance. Premièrement les processus systèmes sont des programmes indépendants, planifiés et exécutés par le système d'exploitation. Un processus peut ensuite exécuter et ordonner des tâches concurrentes en son sein appelé *Thread*. Les *Threads* exploitent un contexte et une zone de mémoire partagée pour les échanges, à l'inverse des Inter processus call (IPC) pour les processus. Les *Threads* accèdent de manière concurrente à la même zone mémoire et doivent exploiter des mécanismes de synchronisation afin de se coordonner tels que les : rendez-vous [Cha87], [RAY12], sémaphores [Hoa74], [Dij01], [Dij71] et autres verrous [CHP71], etc...

Exploité en marge de la programmation orientée objet actuel, le développement à base de *Threads* a été très décrié [Lee06b],[Sut05] car forçant le développeur à expliciter les points de synchronisation de façon optimale sans quoi les performances s'en trouvent très limitées. Cette synchronisation est d'autant plus indispensable qu'aucun mécanisme ne permet d'échange hormis la zone mémoire partagée.

Apparus dès les années 85 [Agh85] mais remis sur le devant de la scène dans des langages modernes comme Scala [HO09], le paradigme d'*acteur* vise à fournir une solution pour rapprocher le mécanisme de synchronisation du modèle évènementiel et des objets. De façon très schématique le paradigme d'*acteur* se base sur la notion d'objets communicants échangeant des messages asynchrones. Un modèle évènementiel remplace donc les IPC et le modèle de concurrence est assuré par des files d'attente où sont systématiquement déposés tous les messages au moment des envois. Virtuellement un *acteur* est donc un processus léger qui est en réalité ordonnancé sur un ou plusieurs *Thread* réel(s). Plus proches du modèle à objet les *acteurs* permettent de virtuellement organiser le programme comme des objets parallèles communicants possédant un état et une zone mémoire indépendants.

De manière naturelle, les *acteurs* ont été étendus pour envoyer des messages non plus uniquement localement mais également à distance. Le modèle de concurrence par évènement est donc plus facilement transposable pour l'informatique distribuée. D'ailleurs les implantations réseaux de *Socket* actuelles simulent également un modèle évènementiel [BCS06]. A l'inverse les modèles d'architecture multicoeurs haute performance s'inspirent également de ce modèle réseau. Ce principe de zone mémoire individuelle a même

11. http ://www.erlang.org/

été récemment étendu pour la construction de puces multicoeurs, ainsi le projet *Single-Chip Cloud Computer d'Intel* exploite ce type d'architecture distribué directement dans une puce [VdWMH11],[PSMR11].

2.3.7 Patron et style d'architecture

Les architectures distribuées et notamment les architectures asynchrones partagent un ensemble de problèmes communs, notamment autour de la synchronisation de flux (séquence de messages) de données. A la manière dont le GOF [Gam95] a classifié les différentes solutions possibles pour résoudre les problèmes communs de la programmation orientée objet, Schmidt *et al* [SSRB00] classifient dans un ensemble de patrons les solutions pour la définition d'architectures complexes. Ces patrons définissent alors des solutions réutilisables en définissant la classe de problème générique auxquels ils apportent structurellement une réponse. Par exemple des problèmes tels que la corrélation de messages de plusieurs sources et le (de)multiplexage de messages asynchrones pour un déclenchement de tâche synchrone y sont abordés. Schmidt *et al* poussent même l'application de ces patrons en dehors des frontières des systèmes informatiques puisqu'ils trouvent leurs applications dans des systèmes réels tels que le système d'appel d'Ericsson® ou le système de suivi de FedEx®. A l'issue de cette analyse un langage patron est même proposé afin de définir les relations entre ces patrons de conception d'architecture. Peu après cette analyse a été publié par Hohpe *et al* un livre nommé Enterprise Integration Pattern (EIP) [HW04]. Ce document détaille et raffine les patrons d'architecture en les spécialisant pour la gestion des flux entre systèmes d'informations différents. Hohpe *et al* détaillent dans ce document une notation qui permet d'assembler ces patrons pour la modélisation des flux à la manière du langage de Schmidt *et al* [SSRB00] . Par nature asynchrone, et de forte charge les échanges entre les systèmes d'information ont été la motivation pour le développement de middlewares spécialisés : d'abord les Enterprise Application Integration (EAI) centralisés puis les Enterprise Service Bus (ESB) décentralisés. Les patrons d'architecture de Schmidt *et al* et de Hohpe *et al* ont été largement exploités pour la construction de ces bus spécialisés. Le projet Apache Camel [12] est même une implantation directe de la notation Hohpe *et al* et est toujours activement exploité par de nombreux projets.

2.3.8 Algorithmes de consensus

Les algorithmes de consensus ont été largement exploités en informatique distribuée depuis deux décennies. Initialement prévu pour les bases de données réparties [Tho79] un algorithme de consensus vise à garder cohérente une donnée répliquée et modifiée par des processus concurrents. Afin de garantir une cohérence globale des répliquats, chaque demande de mise à jour doit obtenir un consensus majoritaire avant d'être appliquée localement. A l'inverse des algorithmes qui laissent diverger les différentes copies d'une donnée, les algorithmes de consensus imposent donc un ordre total des valeurs successives prises par cette donnée. En d'autre termes, la donnée répliquée sur chaque noeud

12. http ://camel.apache.org/

suit la même séquence des différentes valeurs successives. Le but est de synchroniser et de coordonner les valeurs de cette donnée afin d'assurer suffisamment de cohérence pour sa sauvegarde, et ce même en perdant des noeuds en cours de fonctionnement. Cette approche a fait son apparition dans les années 85 [FLP85]. Elle a été rapidement suivie par la définition des algorithmes Paxos qui visent à définir une résistance à la perte de plusieurs noeuds. Une explication de ce procédé est celle de Leslie Lamport [Lam01] qui définit les différents rôles et leurs responsabilités dans cette classe de protocole. De façon simplifiée, un algorithme de Paxos repose sur la collecte de promesses d'acceptation des différents "acceptors" afin d'évaluer le nombre de noeuds allant accepter la nouvelle valeur. Un ensemble de noeuds faisant partie de l'ensemble des répliquats disponibles est alors appelé un Quorum : si ce quorum est suffisamment grand pour assurer la majorité absolue, le consensus est accepté dans le cas contraire il est rejeté et la mise à jour est annulée.

2.3.9 Algorithmes épidémiques

L'ordre total assuré par les algorithmes de consensus impose de nombreuses restrictions sur les évolutions de la donnée protégée mais également sur les topologies réseaux mises en oeuvre. Implicitement son fonctionnement n'est optimal que lorsque les communications sont directes entre les "acceptors" pour pouvoir exploiter des services de multicasts réseaux afin de limiter la charge des échanges. Implicitement également chaque "leader" doit connaître la totalité de la topologie réseau pour prendre la décision ou non de garantir le consensus.

Avec l'avènement des architectures très large échelle que l'on peut trouver dans les applications d'échanges de données ou encore avec les réseaux sociaux, s'est développée une branche en opposition au consensus qui visent à n'assurer qu'un ordre partiel. Ces algorithmes que l'on peut classifier majoritairement sous le terme *Gossip* exploitent un mode de propagation épidémique calqué sur la propagation virale du monde vivant. Ce terme fait également allusion à la façon dont une "rumeur" peut se propager dans les réseaux sociaux. L'idée minimaliste des algorithmes de Gossip est d'effectuer une communication directe entre les noeuds d'un réseau pair-à-pair afin de synchroniser la valeur d'une donnée. La communication est initiée périodiquement par un noeud avec un de ses voisins choisi aléatoirement. Ce type d'approche est donc très adapté aux topologies maillées pair-à-pair où les noeuds ont des connexions fortement intermittentes. En effet de part son caractère opportuniste et sa totale décentralisation ce type de dissémination offre une forte résistance à la perte de noeud pendant le fonctionnement.

Ce type d'approche a été exploité afin de construire des protocoles de diffusion sur des réseaux pairs-à-pairs [RHP+03], ou des constructions opportunistes de topologie par découverte de proche en proche comme dans le projet T-man [JB06a],[Mon08].

Le choix d'un voisin par un noeud nécessite cependant une couche de réflexion sur la topologie du réseau. Cette couche de réflexion est très complexe voire impossible à maintenir de façon exhaustive sur des réseaux de grande taille [JK06]. Afin d'assurer un passage à l'échelle, plusieurs techniques permettent de distribuer cette information en sous-partie (*slicing*) sur chaque noeud [JVG+07]. Le caractère totalement décentralisé

des protocoles Gossip les rend très tolérants à cette vision divergente et partielle de l'ensemble des noeuds du système.

2.3.10 Cohérence à terme

Depuis les années 90, les bases de données sont au cœur d'applications mondialement distribuées. Le partitionnement et la réplication des données sur différents serveurs permettent alors de répondre et de lisser la charge sur ces bases de données. La gestion de la cohérence des différents répliquats des entrées de la base se confronte alors aux problèmes qui ont motivé les algorithmes détaillés en sous-section 2.3.8 et sous-section 2.3.9.

L'assurance d'unicité d'une entrée de la base, et donc de consensus sur tous les repliquats d'une telle application est d'après Fischer [FLP85] *et al* extrêmement coûteuse. De manière analogue aux algorithmes *gossip*, le modèle de cohérence à terme (*eventual consistency*) est proposé afin de rendre la réplication extensible (*scalable*). Dans ce nouveau modèle chaque entrée de la base peut diverger et prendre une valeur différente sur chaque répliquat, introduisant une incohérence potentiellement conflictuelle des données. Le modèle de cohérence à terme n'assure donc pas à un instant T la cohérence d'une valeur sur un répliquat mais permet d'assurer la convergence des répliquats vers une valeur commune à un instant T+n. Terry *et al.* illustrent l'usage de ce modèle de divergence des bases dans Bayou [TTP+95] qui détaille une approche dédiée à la réplication de données au-dessus de réseaux intermittents et donc naturellement sujets à la divergence.

Ce modèle de cohérence à terme remet en cause le modèle Atomicity, Consistency, Isolation and Durability (ACID) [Pri08] qui permettait aux bases de données de garantir l'unicité et la cohérence des données entre deux transactions. Cette rupture avec le modèle ACID a été formalisée sous la forme du théorème CAP de Brewer *et al* [Bre00],[GL02]. Ce théorème annonce qu'il est impossible d'assurer en même temps la cohérence, la disponibilité et la résistance à la perte d'une partition des données. L'application de ce théorème et de la cohérence a donné lieu à de nouveaux modèles de bases de données répliquées, tels celui proposé par Gustavsson *et al* [GA02] ou encore le modèle *BASE* de Pritchett [Pri08] *et al* lié à l'architecture d'Ebay®.

Si le modèle de cohérence à terme permet de garder des performances d'insertion *scalable*, la garantie de convergence des bases de données est un problème difficile lié à la capacité des répliquats à résoudre les conflits de version. En effet pour gérer les conflits, les approches telles que Bayou [TTP+95] diffusent les différentes évolutions de valeurs afin de laisser à chaque répliquat la possibilité de résolution en reconstruisant l'ordre total des mises à jour. Cette *conscience* de l'ordre des mises à jour et donc de l'état de convergence nécessite alors un accord des répliquats sur l'ordre pour la résolution, ce qui freine l'extensibilité (*scalability*) d'une telle solution. En réponse à ce problème, Baldoni [BGL+06] *et al.* proposent un modèle de cohérence à terme sans connaissance de l'état de convergence. La convergence des répliquats est alors assurée par un algorithme *gossip* de diffusion ainsi qu'une résolution locale des conflits qui effectue un *rollback* en cas de réception non ordonnée. Cette approche permet de garder

le protocole de réplication totalement asynchrone et évite ainsi tout usage de consensus entre les répliquats, assurant ainsi une extensibilité horizontale.

Chapitre 3

Etat de l'art

3.1 Système autonomique et boucle d'adaptation

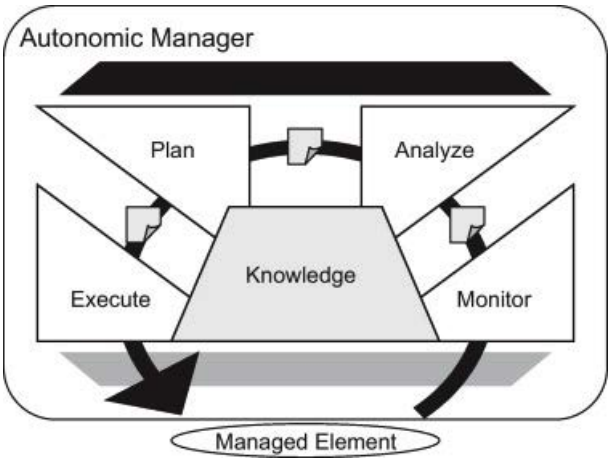
Les systèmes adaptatifs peuvent pendre plusieurs formes suivant les objectifs et les origines des adaptations. Leurs variantes nommées systèmes autonomiques visant l'adaptation pour l'auto-réparation, ont été parmi les premières à formaliser une boucle de rétroaction nommée Monitor Analyze Process Execute (MAPE) englobant la prise de décision et l'application de l'adaptation. Ce processus détaillé en sous-section 3.1 a été par la suite repris dans de très nombreux travaux autour des DAS. Weyns *et al* [WSG+12] base d'ailleurs d'un ensemble de patrons de conception sur cette abstraction pour expliciter les coordinations d'adaptations multinoeuds. La suite de cette section 3.1.1 donne un autre exemple de coordination de boucle de réparation dans le monde des agents avec l'exemple du framework Akka qui adopte une organisation hiérarchique pour coordoner les prises de décision. Un autre exemple de cette coordination est donné en 3.1.2 dans le monde des approches à composants dont la plupart exploitent des superviseurs centralisés pour réaliser la boucle d'auto-réparation. De manière synthétique, cette section rappelle les concepts des boucles d'adaptation des systèmes autonomiques ainsi que les possibilités pour leurs coordinations dans un environnement multinoeuds afin d'en extraire les besoins pour les DDAS.

Boucle autonomique d'adaptation

Entre 2001 et 2003, IBM Research a publié plusieurs articles [GC03],[KC03] et livres blancs sur sa vision de l'informatique autonomique. Partant du constat que la complexité de gestion manuelle des infrastructures a atteint une taille critique, Ganek *et al* envisagent la construction de système autonomique c'est à dire capable de prendre seul les décisions de maintenance visant à maintenir ses fonctionnalités. Dans la citation suivante tirée d'une de ces publications, il est défini comme pré-requis que ce type de système doit se connaître lui-même et doit donc se construire un modèle de réflexion qui représente ses éléments constituants :

> "To be autonomic, a system needs to "know itself" and consist of components that also possess a system identity."

FIGURE 3.1 – Boucle de contrôle défini par IBM



De même un système autonomique est défini comme fondamentalement dynamique car devant faire face à des états non prévisibles par son concepteur initial :

> "An autonomic system must configure and reconfigure itself under varying and unpredictable conditions."

Kephart, Ganek and al ont extrait alors quatre propriétés qui définissent un système autonomique.

- Self-configuring : le système doit lui-même être capable de définir et composer son architecture en fonction des variations de sa plate-forme d'exécution et de son environnement.
- Self-healing : le système doit être capable de diagnostiquer un problème interne de fonctionnement.
- Self-optimizing : le système doit être capable d'adapter ses ressources disponibles pour optimiser son fonctionnement.
- Self-protecting : le système doit détecter et se protéger des attaques et accès extérieurs.

Ces travaux ont abouti à la définition d'une boucle qui définit le processus standard d'un système autonomique. Nommée MAPE et illustrée sur la figure 3.1 cette boucle préconise une analyse permanente de l'état, une analyse et traitement des résultats aboutissant à un processus devant adapter le système concret.

Cette vision des systèmes autonomiques a inspiré de nombreux travaux qui ont suivi autour des approches Model@Runtime et Agents : en effet l'état réflexif est le coeur de cette vision sur laquelle viennent se greffer les boucles de raisonnement. Inspiré des théories du contrôle, le processus MAPE devient implicitement un maître du système devant seul décider de son évolution. Au delà d'un simple DAS la vision de Ganek and al vise à embarquer non seulement l'état du système mais surtout les processus qui ont permis sa construction, son optimisation et son déploiement. En extrapolant ce concept pour les systèmes distribués il est alors nécessaire soit d'assurer l'existence d'un seul

processus MAPE soit d'assurer la cohésion et synchronisation des multiples MAPE mais ceci n'est pas couvert pas ces travaux.

3.1.1 Akka : Exploiter les erreurs dans le design (Fault as first class entity)

Plus récemment les travaux autour du projet Akka [GSTV11] visent à fournir des constructions au niveau langage pour les problèmes de "Self-healing" et "Self-optimizing". Ainsi, construit autour du paradigme d'acteur 2.3.6, ce framework pour les systèmes distribués poursuit l'idée qu'afin de fournir ces deux propriétés au système il faut non plus exploiter une programmation défensive qui résout les problèmes localement mais au contraire laisser les problèmes se propager jusqu'à une entité responsable du traitement des erreurs. Cette propagation se fait via une hiérarchie d'Acteur se supervisant et définissant des stratégies de réparation. Ce framework n'est pas construit pour les systèmes dynamiques car les possibilités d'adaptation sont directement implantées dans le code et ne visent pas le chargement à chaud de nouvelles stratégies. Néanmoins les travaux de Ghosh and al montrent que les paradigmes d'acteurs permettent aux développeurs d'écrire de façon performante de nombreux cas d'usage du distribué en gérant explicitement les accès concurrents. Mais surtout leurs résultats [GSTV11], [GSTV12] montrent que la gestion du cycle de vie de ces entités permet de déployer les politiques de "Self-Healing" sans avoir connaissance du type des acteurs modifiés. Akka adopte donc une sous-partie du cycle MAPE : analyse et instanciation, qu'il déploie dans plusieurs entités nommées superviseurs. L'organisation hiérarchique de ces superviseurs est donc un premier pas vers la composition des MAPE. L'introduction dans le langage de développement des primitives de contrôle du sous-ensemble de MAPE permet de rapprocher les algorithmes de prise de décision des points de détection de fautes (émission d'exception en langage Java par exemple). Cependant le couplage fort entre les détections de fautes, les primitives d'adaptation et les plates-formes contraint le système à prendre des décisions à application immédiate avec le risque d'irréversibilité que cela comporte.

3.1.2 Du design au runtime : chargement à chaud et déploiement

La vision d'IBM sur les systèmes autonomiques était inspirée par le développement de manager d'infrastructures et c'est dans ce domaine que se trouve la plupart des implantations de la propriété "Self-configuring". En effet pour pouvoir adapter un système et déployer ses mises à jour, il est nécessaire de relier le modèle de design qui sert à construire d'une manière ou d'une autre la configuration à un modèle de déploiement qui permet à chaud de modifier la plate-forme avec les implantations concrètes. On retrouve cette application de la dernière étape du processus MAPE dans des récents projets de gestion d'ESB tels que Petals Master [1] ou même dans la spécification de la norme Java Business Integration (JBI). Ces projets visent à déployer les assemblages de composants formant les bus d'intégration des systèmes d'information d'entreprise.

1. http ://petalsmaster.ow2.org/

Ils exploitent des frameworks tels qu'OSGi [All03] pour effectuer le chargement à chaud d'artefacts de développement (ici : des fichiers JAR). Si ces approches permettent la construction de DDAS là encore il existe un couplage fort entre les décisions du *manager* d'infrastructure et les plates-formes. Ce couplage complexifie l'usage de plusieurs *managers* mais surtout limite son usage avec des connections intermittentes. Il faut cependant retenir de ces projets que le rapprochement de la couche de déploiement (ex : JAR en Java) avec la couche de design (ex : composants) est nécessaire pour pouvoir déployer directement par assemblages de systèmes.

3.2 Critères d'évaluation

D'après la vision autonomique d'IBM et les caractéristiques des DAS actuels un framework permettant de développer des DDAS hétérogènes et continus pour réseaux fiables et non fiables doit comporter les propriétés suivantes :

Modèle de concurrence explicite : le modèle de concurrence doit être explicite dans le modèle de développement pour deux raisons. Premièrement un système distribué doit gérer efficacement les accès concurrents des différents clients sur les points serveurs. Les résultats de Gosh and al [GSTV11], [GSTV12] sur Akka montrent que la définition explicite des points de concurrence est plus efficace pour les développeurs qu'un masquage comme dans les RPC. Deuxièmement, en rendant explicite la concurrence et les points d'accès, le système DDAS peut alors déterminer le meilleur moment pour couper l'exécution d'une entité pour sa mise à jour. Par exemple dans le cas d'un composant, avec un modèle de file d'attente sur les ports, on peut connaitre à quel moment les messages entrants sont consommés et donc à quel moment on peut couper le composant.

Distribué : Distribué par nature un DDAS doit expliciter dans le modèle de design ses noeuds ainsi que leurs relations (c'est à dire son modèle de topologie). Ainsi à la manière des superviseurs d'infrastructure il est possible d'exploiter ce modèle pour déployer et monitorer les noeuds du réseau.

Séparation entre composant métier et de communication : Le cycle de vie des composants métiers et de leurs connecteurs évoluent à des rythmes de vie très différents. Dans les modèles à composants ainsi quand dans les spécifications telles que JBI, cette séparation a déjà été identifiée pour maximiser la réutilisation de composant métier et surtout pour ne pas polluer le code avec des préoccupations différentes. Dans un contexte DDAS cette caractéristique est accrue ; en effet l'adaptation de ces systèmes vise à répartir les composants métiers et à changer leurs mode de communication pour s'adapter au contexte. Cette séparation est donc essentielle pour rendre réutilisables les composants métiers.

Intégration de la boucle MAPE dans le modèle de développement et désign : Afin d'offrir la capacité aux développeurs d'écrire les algorithmes d'auto-adaptation il est nécessaire d'offrir au niveau modèle de développement un accès aux API de la MAPE. Cette API doit couvrir les accès à la couche de réflexion mais également les accès à la couche d'intercession (modification de composant déployé par

exemple). L'intégration de cette API doit permettre de modifier le système courant et ceci inclut également la capacité à déployer à chaud de nouvelles entités. La couche de design offert dans l'API MAPE doit donc d'une manière ou d'une autre garantir l'accès au modèle et aux informations nécessaires au déploiement (Par exemple adresse de serveur de binaire et primitive d'installation local).

Désynchronisation entre le modèle réflexif et d'intersession et la plate-forme : Pour garantir la capacité du système à effectuer des tests sur les adaptations avant déploiement, ainsi que pour lui permettre de prendre des décisions à application non immédiate il est nécessaire que le système ait la capacité de désynchroniser la couche réflexive de la plate-forme.

Hétérogénéité : Le système DDAS devant gérer et adapter des nœuds de type hétérogène il est nécessaire que cette hétérogénéité soit remontée dans le modèle de conception pour sa prise en compte par les algorithmes de décision d'adaptation. De plus cette hétérogénéité doit modéliser les capacités d'adaptation de chaque type de nœud. Par exemple un nœud de type Java est capable de faire du chargement de code à chaud, alors qu'un micro-contrôleur nécessite pour cela une opération de *flashage* de sa mémoire. De ce fait leurs représentations dans le modèle doivent tenir compte de cette différence de capacité et de coût pour réaliser les adaptations.

Dissémination des adaptations : Un système DDAS doit être capable de disséminer les adaptations décidées par un noeud de contrôle vers les noeuds de calcul. Cette propriété est liée à la capacité de désynchronisation dans le cas de transport long ou sur réseau non fiable.

Le tableau suivant présente les acronymes de ces propriétés utilisées par la suite.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt

Les symboles suivants seront exploités pour classifier les travaux existants.

- $+$: Pleinement supporté
- $\otimes$: Non supporté
- $\pm$: Partiellement supporté

La section suivante ne détaille pas seulement des projets ayant pour but la construction de DDAS mais au contraire s'ouvre à des approches spécialisées dans l'usage de noeuds distribués par exemple. De fait, la plupart d'entres elles ne s'intéressent qu'à un pendant du problème décrit ici et n'implantent alors que les propriétés du tableau correspondantes, cette répartition sera analysée en conclusion de cet état de l'art.

3.3 Approches permettant l'Introspection et l'Intercession

Les systèmes adaptatifs se rapprochent des langages réflexifs de part leurs usages, pour l'application des modifications, d'un mécanisme d'intercession. Celui-ci s'accompagne le plus souvent d'un mécanisme d'introspection pour la prise de décision. Cette section se focalise donc sur les approches permettant de construire un système adaptatif en implantant l'une ou l'autre de ces propriétés. Les différents courants des systèmes

adaptatifs, que ce soit les approches à composants ou agents se focalisent sur des points différents pour l'implantation de ces propriétés. Ainsi la première sous-section étudie les approches à composants fournissant une approche très structurelle à ces deux mécanismes qui se focalise sur la réutilisation du logiciel et la définition d'outils de manipulation de ces architectures. La deuxième sous-section s'intéresse aux approches à agents et acteurs qui se focalisent sur les modèles de concurrences et surtout la construction d'une couche de réflexion orientée donnée. Enfin la dernière sous-section détaille les approches M@R qui s'intéressent à fournir ces deux services au travers d'une représentation modèle.

3.3.1 Approches basées sur la notion de composant

Cette sous-section étudie les approches exploitant une notion de composant logiciel pour la construction de la couche d'intercession et d'introspection. Après un paragraphe sur les éléments de classification, cette sous-section détaille tout d'abord les modèles à composants focalisés sur l'expression d'adaptations localisées, puis à partir du modèle OSGi sur les approches ayant rapproché la notion de service et port et enfin à partir du modèle SAM sur les approches abordant le problème de distribution des composants sur différents noeuds.

3.3.1.1 Une classification difficile et multiple

De nombreux modèles de composants ont été développés dans la recherche académique et l'industrie depuis une vingtaine d'années. Cette multitude d'approches est due en grande partie à la multitude des cas d'usage et applications du paradigme, faisant évoluer toutes les approches vers des abstractions spécialisées. Dans un papier de 2011 [CCSV07], Crnkovic *et al* expliquent les raisons de cette diversité et établissent une classification des approches à composants. Pour Crnkovic la diversité vient du flou autour de la définition même de la notion de composant logiciel. En effet dans la définition de Szyperski [SGM02] il est défini : *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies"*. En résumé un composant y est ici décrit comme une notion essentiellement de conception dédiée à la réutilisation de pièce logiciel. Dans une autre définition, Heinman [HC01] *et al* décrivent un composant comme : "A component model defines a set of standards for component implementation, naming interoperability, customization, composition, evolution and deployment". Dans cette deuxième proposition, l'accent est donc mis sur le lien avec un modèle de développement exploitant non plus le modèle comme une notion permettant la composition abstraite du système mais allant jusqu'à la composition des pièces pour le déploiement du système global. Dans l'analyse de Crnkovic *et al* mais également dans celle de Medvidovic qui la précède [MT00], les orientations de chaque modèle expliquent les différences contrastées entre les Architecture Description Language (ADL) qui accompagnent ces modèles. En tant qu'outils les ADL se doivent en plus d'être proches des besoins de l'utilisateur (développeur) et donc se spécialisent chacun à cet effet.

Toujours dans sa classification Crnkovic établit un vocabulaire commun pour le cycle

de vie d'un composant qui va de sa définition d'exigence à son exécution en passant par son déploiement sur des serveurs tiers. Les besoins des modèles CBSE et des DAS ne sont pas totalement alignés mais de nombreux points de cette classification peuvent être repris pour l'établissement de cet état de l'art. Crnkovic *et al* établissent par exemple deux catégories de liens (binding) entre composants : endogène et exogène qui définissent la sémantique de communication, ou la délèguent à la notion de connecteur et de plus ces communications peuvent être asynchrones ou non avec plusieurs styles d'interaction (Aller-Retour ou Aller seulement ou encore rendez vous dans le modèle BIP [BBS06]). Ces critères correspondent donc à la séparation entre composants métier et de communication nécessaires pour la mise à jour des DDAS mais également à la notion d'hétérogénéité d'interaction et mode de communication. Dans un autre critère Crnkovic définit la nécessité de découplage entre langage de modélisation et celui d'implantation, ce découplage entre ADL et plate-forme permet un premier niveau du découplage modèle et composant nécessaire pour les DAS.

En reprenant la conclusion de Crnkovic, aucune approche ne peut résoudre l'ensemble des problèmes intervenant dans la définition d'un tel modèle de composant, l'acceptation et la définition de l'hétérogénéité dans la définition même des types de composant sont donc une réponse pragmatique à cette hétérogénéité d'usage. Enfin cette classification ne traite pas des éléments nécessaires à la construction d'un DAS à l'aide de ces modèles. Est ainsi non décrite l'implication des interactions avec les mises à jour ou le lien avec le processus MAPE nécessaire pour l'interaction entre l'ADL et les plates-formes. Cet usage encore différent et ayant donné lieu à de nouveaux modèles justifie cette étude d'état de l'art vis-à-vis des besoins spécifiques des DDAS qui commence avec l'étude suivante du modèle Fractal 3.3.1.2.

3.3.1.2 Fractal

Le projet Fractal [BCL$^+$06],[DL$^+$06],[LLC07] définit un modèle de composant à usage généraliste, modulaire et extensible pour la définition, le design et le déploiement d'architecture à composants sur différents types d'application. Dans cette approche la vision de Crnkovic et Medvidovic y est abstraite et synthétisée pour offrir un modèle suffisamment générique pour couvrir les besoins d'encapsulation sous forme de boîte noire des composants logiciels capables de communications via des interfaces. Dans le modèle Fractal, les composants sont donc définis par leurs interfaces d'entrées et sorties en équivalence aux ports de la définition de Szyperski. Ce modèle est également hiérarchique, chaque composant peut être lui-même composé d'autres composants fils. Ainsi est proposée une notion de composant composite permettant d'abstraire un ensemble de composants formant un fragment d'architecture sous la forme d'une seule entité qui peut alors promouvoir les ports de ses composants internes vers l'interface du fragment. Une des caractéristiques importantes du modèle Fractal est sa séparation de la notion de composant en deux parties distinctes :

- **un contenu** qui définit l'intérieur d'un composant et est composé soit de code exécutable, soit d'autres composants dans le cas d'un composite.
- **une membrane ou contrôleur** qui définit l'enveloppe extérieure d'un compo-

sant. Elle fournit les fonctions d'introspection et permet d'ajouter de la sémantique d'échange entre les interfaces extérieures et le ou les composants internes. C'est également par elle qu'il faut passer pour effectuer une mise à jour de composant.

Le modèle Fractal est donc complètement hiérarchique et en tire d'ailleurs son nom, les contrôleurs se composent entre eux pour faire de la délégation, à la fois pour propager les adaptations mais également pour les échanges de données. L'assemblage se fait donc entre des composants primaires ou des composants composites via leurs interfaces qui sont reliées via une notion de connecteurs ou liaisons. Six contrôleurs sont proposés nativement dans le modèle Fractal pour gérer différents niveaux d'adaptations : contrôleur d'attribut et de nom (adaptation paramétrique), contrôleur de connecteur et liaison, contrôleur de contenu (adaptation de composite), contrôleur de cycle de vie(arrêt et démarrage), contrôleur de super conteneur (propagation d'adaptation vers le super conteneur). Ce modèle de contrôleur est extensible et permet ainsi de définir de nouveaux niveaux d'adaptation. Fractal propose plusieurs implantations de plate-forme respectant ce modèle abstrait, chacune ayant des spécificités pour le modèle de développement ou les capacités de développement des contrôleurs. *Cecilia* et *Think* sont les implantations C/C++ tandis que FracNet et FracTalk sont respectivement les implantations de référence pour la plate-forme .Net et SmalTalk. Pour la machine virtuelle Java, l'implantation de référence est le projet *Julia* mais un autre nommé *AOKell* [SPDC06] est disponible, cette différence s'explique notamment par leurs capacités de composition des contrôleurs. *Julia* exploite un opérateur de Mixin [BC90], [SB98] tandis que AOKell exploite un approche par tissage d'aspect [KHH+01]. Plus récemment MIND [2] accompagné du consortium OW2 vise à fournir une implantation de plate-forme dédiée pour le monde embarqué.

Léger *et al* [LLC07] ont également étendu ce modèle avec un langage nommé *FScript* [3] dédié à piloter une plateforme à composant Fractal avec un langage de script dont l'exécution est transactionnelle et respecte les propriétés ACID. Ainsi l'adaptation pilotée par un tel langage permet de faire de la récupération d'erreur et permet ainsi de garantir les transitions du système d'un état sain vers un autre. Ce mode transactionnel est réalisé à l'aide d'un verrou et d'un historique d'état inversable qui permet de revenir à un état initial en cas d'erreur.

Le modèle Fractal offre donc une couche de réflexion du système en fonctionnement, cependant ce modèle est l'exacte représentation de la réalité et n'est pas désynchronisable de la plate-forme. Ainsi toute modification du modèle implique immédiatement une modification de la plate-forme, d'où les travaux qui ont été réalisés pour rendre cette opération transactionnelle. Ainsi il est impossible d'exploiter le modèle en amont de ses modifications soit dans une phase exploratoire soit dans une phase de synchronisation de plate-forme. Fractal ne définit d'ailleurs par de notion de plate-forme et ne peut donc pas définir un système DDAS, par contre ses multiples implantations de plate-forme respectant la spécification permettent de gérer un premier niveau d'hétérogénéité. Une autre limitation vis-à-vis des besoins exprimés pour la construction DDAS est le manque

2. http ://mind.ow2.org
3. http ://fractal.ow2.org/fscript/

de capacité de déploiement à chaud d'un nouveau composant non initialement prévu au design, ce manque interdit par transitivité tout déploiement et design continu. Le modèle de concurrent peut lui par contre être inclus dans le modèle de contrôleur et enfin le lien avec le modèle de développement peut être explicite avec l'utilisation de projet tiers tel que *Fraclet* qui permet de décorer des classes Java avec des annotations pour extraire la représentation composant.

En synthèse le modèle Fractal a largement contribué à poser les jalons des architectures dynamiques à composants, cette base a d'ailleurs été reprise dans des travaux plus récents tels que Frascati, détaillés ci-dessous.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	⊗	±	+	⊗	⊗	±	⊗

3.3.1.3 Standard SCA / Projet FraSCAti

Le standard Service Component Architecture (SCA) [4] [BADI05] [MR10] proposé par l'OSOA (Open Service Oriented Architecture) est issu d'une volonté de rapprocher les architectures à service et à composant. De cette initiative commune de plusieurs compagnies telles que IBM, Oracle ou SAP est issu un modèle permettant à la fois d'exprimer la notion de service logiciel mais également composants ainsi que leurs assemblages pour modéliser une architecture complète. Toutes les interconnexions entre composants suivent donc le paradigme service qui sert également à typer les contrats de composants. Les implantations les plus connues de cette norme sont les projets Apache Tuscany [5], Newton [6] (au dessus d'OSGi) ou encore FraSCAti (au dessus de Fractal).

Le modèle SCA est définit autour de 4 grands principes :

- Indépendance du langage de programmation
- Indépendance des IDL (*Interface Description Language*)
- Indépendance des protocoles de communication
- Indépendance des propriétés non fonctionnelles

Le modèle SCA est donc abstrait et peut encapsuler de manière homogène (avec le même modèle) des composants écrits dans des langages et définis avec des IDL. Les implantations réelles des connecteurs qui relient les ports des composants peuvent être réalisés à l'aide de WebService, RPC ou échange de message par bus, mais dans tous les cas l'exposition du service sera homogène en respectant le modèle SCA.

Le standard SCA définit donc essentiellement la définition de composant, service et architecture mais ne spécifie pas comment celui-ci doit être exploité pour la réalisation d'adaptation dynamique dans le cas des DAS. Le standard ne définit donc pas d'information quand à la migration d'état de système et ceci est laissé à la charge des implantations de plate-forme qui doivent résoudre ce manque, comme c'est le cas avec le projet FraSCAti.

4. http ://www.oasis-opencsa.org/
5. http ://tuscany.apache.org
6. http ://newton.codecauldron.org

Plate-forme FraSCAti La plate-forme FraSCAti [SMF+09a],[MMR+10],[SMR+11] est une implantation du standard SCA au dessus d'une plate-forme Fractal, profitant ainsi des capacités de reconfiguration de ce dernier. Classifié sous les plates-formes exogènes FraSCAti injecte donc la couche de transport concrète (message via JMS, webservice RPC, appel de méthode Java) suivant le type de liaison définie dans le modèle. L'architecture même de la plate-forme est construite à l'aide d'une approche SPL (*Software Product Line*) qui permet de construire une plate-forme d'hébergement de composants à l'aide d'assemblage de composants elle-même. Cet assemblage peut alors choisir plusieurs points de variations tels que les types de couches de transport gérés au moment des liaisons permettant ainsi d'être extensible sur les capacités de communications entre composants.

FraSCAti est donc défini pour la gestion des DAS et définit du fait de nombreux points nécessaires pour la gestion des DDAS. Le lien avec le modèle de programmation est fait par le biais du modèle SCA tandis que les contrôleurs Fractal apportent l'API d'accès au cycle MAPE permettant le déploiement de composants. La plate-forme ne permet cependant pas le chargement à chaud, ce qui interdit le design continu des composants. Le modèle de concurrence des ports n'est pas inclus dans le modèle de développement mais peut être explicité grâce à des connecteurs particuliers entre composants isolant ainsi les processus. Cette notion de connecteur abstrait permet également d'encapsuler des sémantiques de communication hétérogènes, mais qui sont alors cachées derrière une abstraction de service. Cependant tout comme le modèle Fractal, le modèle réflexif de FraSCAti est fortement lié à la plate-forme, il est alors difficile de désynchroniser le modèle du système pour effectuer une recherche exploratoire. Les reconfigurations peuvent cependant exploiter l'usage d'un script tel que FScript pour palier ce manque d'un point de vue dissémination mais pas pour l'usage réflexif.

Le modèle FraSCAti ne définit également pas de notion de noeud ni de capacité de synchronisation entre noeuds. La notion de service du modèle permet cependant la définition de code que l'on peut appeler à distance mais ne permet pas de modéliser la localisation même de l'hébergeur de ce service. Le modèle SCA ne peut de fait que permettre la modélisation d'un seul noeud du DDAS, mais cependant permet la modélisation de ses relations avec l'extérieur (services hébergées sur d'autres noeuds).

Pour synthèse, le modèle SCA définit donc une notion de modèle de composant indépendant du cycle et langage de développement, le principal manque quand à l'utilisation du modèle FraSCAti pour la définition d'un DDAS réside donc dans le manque de modélisation de la topologie, des noeuds et de leurs diversités d'exécution mais surtout de synchronisation. L'application des travaux de cette thèse sur ce type de plate-forme est donc tout à fait possible et fait même partie d'une collaboration avec l'équipe qui le développe, détaillée dans la partie perspective.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
±	⊗	+	+	⊗	⊗	±	⊗

3.3.1.4 ArchJAVA

Les approches à composants s'accompagnent d'outils d'assemblages tels que les IDL. Pour des raisons de séparation de préoccupations, ces outils sont découplés du modèle de développement. Ainsi par exemple pour un modèle de développement s'appuyant sur le langage Java, le modèle d'assemblage peut prendre la forme d'un fichier XML. Le couplage plus ou moins fort entre le modèle de développement et celui de design et d'assemblage assure à la fois la réutilisation mais introduit également un risque d'incohérence : par exemple le nom des classes Java utilisées dans l'IDL peut différer des classes compilées. A l'opposé des autres approches qui traditionnellement permettent aux IDL de pointer par exemple la définition en Java d'une interface, Aldrich *et al* proposent d'inclure le modèle de design de composant directement dans le langage d'implantation, en l'occurrence le langage Java. Ainsi l'approche ArchJava [ACN02],[ACN06] est construite par extension du langage Java dont elle définit un pré-compilateur. Ainsi étendu, le langage Java inclut les concepts des modèles à composants directement dans sa syntaxe textuelle, permettant ainsi dès la phase de développement de définir les caractéristiques composant à extraire du code orienté object.

Cette approche permet rapidement de faire le lien entre le modèle de design et le modèle de développement, mais cependant elle hérite également des limitations du langage hôte : Java. Ainsi cette approche dans le cas des DDAS ne peut donc pas exprimer de modèle désynchronisable de la plate-forme, et permet encore moins d'offrir une couche de modélisation autre que pour les plates-formes Java. Il est cependant possible d'offrir un modèle de concurrence pour les ports au moment de la génération et compilation des composants. Cependant l'inclusion de concept dans le langage rejoint les concepts de DSL interne qui permettent rapidement de décorer et extraire les méta-informations d'un langage hôte. Ce lien avec le modèle de développement est donc à mettre en opposition avec les approches par annotations et dans le cas de notre approche est donc intéressant pour l'extraction du modèle abstrait issu de l'implantation des composants. De plus l'approche par DSL permet aussi de mettre en lumière le besoin d'abstraction proche de l'utilisateur et un modèle de développement cohérent et facile de prise en main pour l'adoption par une communauté de développeurs.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
±	⊗	⊗	⊗	⊗	⊗	⊗	⊗

3.3.1.5 OSGi / DOSGi

OSGi pour Open Service Gateway initiative est une organisation pour la création d'une plate-forme d'exécution dynamique initialement prévue pour les passerelles domotiques. Depuis renommée en *OSGi Alliance*, cette organisation héberge une approche outillée à composant à but généraliste qui se veut l'implantation standard des modules dynamiques pour la plate-forme Java. La spécification OSGi définit donc un framework de gestion de cycle de vie d'un ensemble de composants stockés dans une notion de module nommé *Bundle*. Un *Bundle* désigne un package spécifique issu du JAR des plates-formes Java et obligatoire pour le déploiement. Celui-ci qui peut définir des méta-

informations supplémentaires sous la forme d'un fichier manifeste et ainsi déclarer un contrat de dépendance vers d'autres *Bundles* ou des fragments de *Bundles* en exploitant la notion de package du langage Java. Dans sa forme primaire le framework OSGi définit donc un framewok de module Java, déployable à chaud dont la granularité de dépendance est soit le JAR soit le package de classe Java et dont le cycle de vie s'accompagne de code Java à exécuter (au démarrage et à l'arrêt) défini dans des classes spécifiques nommées *Activator* et représentant le code interne d'un composant. Le framework définit également une notion de service interne, dont l'inscription est dynamique dans un registre central à la plate-forme. Les contrats des composants OSGi sont donc orientés dépendances, mais ont été étendus dans des projets tels que SpringDM [7] ou Blueprint [8] pour définir des contrats avec des dépendances de service plus proches des contrats de composants existants dans d'autres approches telles que Fractal.

Les deux implantations de référence d'OSGi sont les projets Apache Felix [9] et Eclipse Equinox [10]. Le premier est exploité dans de nombreux ESB tandis que le deuxième est la base de l'architecture de l'environnement de développement Eclipse [11]. Une initiative menée par la société IS2T a également abouti à une implantation d'OSGi sur des environnements contraints [12] au-dessus d'une plate-forme JavaME [Keo03]. Dans cette forme initiale la spécification OSGi ne définit pas de couche de modélisation isolée, ni de notion de noeud ni topologie.

Cependant récemment, la spécification a été étendue avec DOSGi [DSH+09] pour prendre en compte la notion de service de distribution. Le principe est le suivant : chaque service exposé dans une plate-forme peut être rendu public en configurant le noeud hébergeant, à l'inverse chaque enregistrement de service peut se faire sur un service à distance en donnant la configuration nécessaire dans le noeud hébergeant (information tel que l'ip, le point d'ancrage, etc ...) . En d'autres termes, les noeuds DOSGi font tous partie d'un bus de service spécifique, soit configurés de manière ad hoc uniquement en point-à-point pour les liens distants nécessaires ou bien de manière plus centralisée à l'aide de projet tiers tel que ZooKeeper [HKJR10]. Vis-à-vis du modèle de développement et du modèle de composant sous-jacent, les liaisons locales et distantes sont donc homogènes et surtout cachées pour le développeur qui n'a pas de fait le choix de la sémantique de communication. Cependant le bus OSGi définit également une notion d'évènement qui est alors distribuable suivant les implantations et peut pallier en partie de ce manque. L'implantation de référence de DOSGi est réalisée à l'aide du projet Apache CXF [13], elle consiste donc à cacher les appels distants sous les RPC des WebServices.

Si l'implantation DOSGi apporte la distribution sur une plate-forme dynamique qui permet alors de réaliser un DAS, elle rentre dans la catégorie des plates-formes exogènes

7. http ://www.springsource.org/osgi
8. http ://aries.apache.org/modules/blueprint.html
9. http ://felix.apache.org/
10. http ://www.eclipse.org/equinox/
11. http ://www.eclipse.org/
12. http ://www.is2t.com/en/products-microej-packs-soa.php
13. http ://cxf.apache.org/dosgi-releases.html

de la classification de Crnkovic *et al.* Ainsi les appels RPC sont cachés derrière un bus de message commun aux plates-formes, interdisant du même coup au modèle de composant et à ses utilisateurs de pouvoir profiter de la diversité des modes de communication inhérentes à la plate-forme et indispensables à la construction d'un DDAS mobile. De plus aucun mécanisme de synchronisation des plates-formes n'est inclus directement, et aucun modèle de déploiement non plus, rendant la configuration du cluster ad hoc et difficile, et encore plus dans le cas d'évolution. Enfin de manière générale, OSGi ne définit pas une sémantique de concurrence sur les ports (services OSGi) puisqu'il adopte la sémantique d'appel de service de Java, il est ainsi difficile de prévoir la réaction de composant en cas de mise à jour des liaisons et encore plus dans le cas distribué. Cependant OSGi apporte de nombreux outils pour la construction d'un DAS mononoeud, il a d'ailleurs été exploité comme cible de plate-forme dans cette thèse, ceci est détaillé dans la section validation.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	±	⊗	+	+	⊗	⊗	⊗

3.3.1.6 iPOJO

iPOJO [EHL07],[EH07],[DBE08] est un projet inclus dans la pile logicielle d'Apache Karaf[14] et dont le but est d'offrir un modèle de composant au dessus du modèle de service de la plate-forme OSGi. Ce projet comble le manque de description de contrat de composant d'OSGi en proposant un rapprochement des paradigmes services et composants. Ainsi les fonctionnalités déployées sur la plate-forme OSGi sont encapsulées dans une notion de composant, dont les ports, décrits sous forme de service, sont liés par extension au mécanisme d'abonnement de service d'OSGi.

Le modèle de composant iPOJO résultant de cette association se focalise sur le développement de plate-forme autonomique pour la plate-forme d'exécution Java. Pour cela, le modèle de développement inclut une notion étendue de résolution de services. Ainsi il est possible d'étendre la façon dont la plate-forme lie les composants les uns aux autres afin de respecter les contraintes définies dans les ports. Cette résolution peut se faire soit localement soit sur des services distants dans le cas d'une plate-forme DOSGi. Ainsi l'assemblage des capacités dynamiques d'OSGi et les résolutions de iPOJO permet de construire des DDAS auto-adaptatifs adaptés à un contexte pervasif.

Tout comme DOSGi le modèle iPOJO ne propose pas de solution pour construire une réflexion désynchronisable. Ainsi toute modification se fait directement sur la plate-forme locale où directement sur une plate-forme distante. iPOJO propose donc à l'inverse une vision autonomique où les noeuds ne connaissent qu'un minimum d'information pour se lier aux services des uns et des autres.

Pour la construction d'un DDAS mobile, cette solution souffre d'un manque d'explicitation des synchronisations des plates-formes et cache les communications sous un mécanisme de RPC qui interdit toute propagation en mode pair-à-pair.

14. http ://felix.apache.org/site/apache-felix-ipojo.html

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	⊗	⊗	+	+	⊗	⊗	⊗

3.3.1.7 Projet SAM / Modèle iPOJO

Le projet SAM est le résultat de thèse d'Eric SIMON [Sim11]. Il y définit une approche généraliste répondant aux besoins spécifiques au domaine de la domotique quand à l'interconnexion de différents éléments et objets communicants hétérogènes. En effet dans ce domaine la multitude de medias et protocoles de communication rend difficile la création d'architecture globale faisant collaborer les différents éléments. La vision de ce projet est donc de déployer une plate-forme colocalisée avec chacun des différents éléments du réseau (par exemple domotique) afin de déployer des composants permettant une communication et une découverte homogène entre eux. Une fois cette couche de médiation déployée, il est alors possible dynamiquement de les lier entre elles par le biais de services distants de façon opportuniste afin de respecter leurs besoins exprimés à l'aide de contrats d'interface.

Pour la réalisation de cette plate-forme SAM, les auteurs (Simon, Estublier *et al* [Sim11]) s'appuient sur un modèle de composant étendu s'appuyant lieu-même sur le projet iPOJO dont ils reprennent la plate-forme d'exécution dynamique OSGi qui permet ainsi la réalisation de l'API MAPE et le déploiement à chaud. La notion de noeud de calcul est présente dans ce projet sous la terminologie d'*Abstract Machine* qui définit alors un conteneur de composant participant à la construction du système distribué global. Chaque plate-forme est donc administrée par un administrateur qui déploie des composants ayant des besoins externes en services. La plate-forme sélectionne alors dans les services environnement déployés sur les autres noeuds SAM un service compatible et le lie au composant local [EDSMG10]. Issus des pratiques autonomiques, ce mécanisme opportuniste permet d'abstraire cette problématique pour le concepteur du composant pour qui les appels distants se résument à des appels de services.

SAM est donc une plate-forme exogène qui masque les appels réseaux et leurs résolutions aux composants. Il n'y est donc pas possible de mixer plusieurs sémantiques de communication entre plates-formes, mais à l'inverse il est possible d'homogénéiser les différents protocoles de communication existants sous le même paradigme service. SAM ne définit pas non plus de modèle désynchronisable permettant à un noeud d'avoir une vision du système pour réaliser les adaptations. A l'inverse chaque noeud ici se retrouve investi de la mission de se lier à des services compatibles et est donc autonome sur sa reconfiguration.

En résumé l'approche proposée par SAM exploite une vision autonomique avec des noeuds indépendants, à l'opposé du besoin de réflexion distant nécessaire pour faire de l'adaptation dans les DDAS. Naturellement la synchronisation de l'état global des noeuds ne peut être offerte puisqu'elle est réalisée par une résolution au niveau des services dans SAM, perdant ainsi toutes possibilités d'avoir des informations au-delà du contrat de service offert. Le modèle de composant proposé est bien lié avec un modèle de développement, issus du projet iPOJO mais ne définit pas de modèle de concurrence, et enfin la plate-forme exogène qui masque les appels distants ne permet pas de modéliser les différentes stratégies de communication d'un DDAS.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	±	⊗	+	+	⊗	±	⊗

3.3.1.8 SOFA 2.0

Le modèle SOFA 2.0 [BHP+07],[BHP06],[HP06] est un projet qui fait suite à la première version du nom et est développé par l'université de Prague. Le modèle SOFA 2.0 est un modèle hiérarchique riche de plusieurs entités de première classe, telle qu'une notion de connecteurs qui encapsulent les différentes sémantiques de communication que peuvent contenir une architecture à composants. Le but de ces entités est de permettre des échanges entre plusieurs entités (en l'occurrence des ports de composants), avec diverses stratégies d'échanges. Par exemple le but est de pouvoir modéliser des stratégies de communication telles qu'on les retrouve dans les bus de communication comme le "Publish and Subcribe". L'approche SOFA vise également les déploiements d'architecture dynamique, le modèle contient donc une notion de répertoire de binaire afin de pouvoir calculer et appliquer les demandes de reconfiguration. Ce répertoire de binaire est la pièce centrale de l'approche de SOFA, elle contient les composants ainsi que les meta-datas nécessaires. Un ensemble de plates-formes appelées SOFAnodes servent de conteneur de composants et sont exploitables via des clients d'architecture qui permettent de faire les demandes de reconfiguration, qui ensuite se connectent au répertoire de binaire pour faire le déploiement concret.

Le modèle SOFA sépare donc le modèle de connecteurs du code métier, mais ne définit pas de modèle de concurrence sur les ports de composants permettant d'isoler les cycles de vie des composants. L'approche ne définit pas non plus de notion de noeud et ne permet pas dans l'ADL de définir une représentation du système multinoeuds complet. Cependant l'approche avec répertoire de dépôt de binaire ainsi que l'accession par client donne lieu à des architectures distribuables. En résumé pour la construction d'un DDAS SOFA n'offre pas de couche de modélisation désynchronisable qui permettrait de faire des adaptations sur différents noeuds sans explicitement devoir s'y connecter, ce qui pose problème dans un contexte sporadique tel que dans les réseaux mobiles ou les communications multi-sauts.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	±	±	+	+	⊗	±	⊗

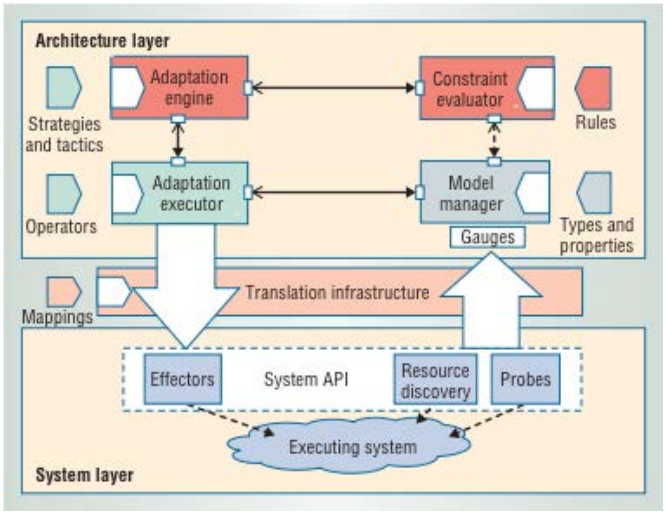
3.3.1.9 Rainbow

Le modèle Rainbow [GCH+04], développé par Garlan *et al* à Carnegie Mellon University a pour objectif de proposer une solution pour la gestion de l'adaptation dans les systèmes distribués complexes. L'idée principale de ce projet est de construire une abstraction de l'architecture du système et d'avoir un certain nombre de stratégies prédéfinies qui permettent d'améliorer et d'adapter ce modèle d'architecture. Basé sur une abstraction, ce modèle d'architecture peut être appliqué à distance sur les différents noeuds pour en reconfigurer l'état, et leur permet alors de faire face à une nouvelle situation telle que la montée en charge du nombre du client.

Rainbow se veut indépendant des langages et plates-formes d'exécution. Une notion de serveur abstrait encapsule donc la notion des noeuds d'exécution capables de déployer des composants. Pour accéder de manière homogène à ses noeuds, Rainbow définit une notion de *RuntimeManager* qui sert de pilote pour dialoguer avec les plates-formes d'exécution. Chacun de ces *RuntimeManager* doit donc offrir les primitives d'accès standard pour un client extérieur voulant reconfigurer le système. Ils doivent également offrir les primitives concrètes d'implantation des adaptations sur leur plate-forme locale qu'ils pilotent. Ainsi l'hétérogénéité des plates-formes d'exécution est prise en forme dans ce projet par l'implantation de multiples *RuntimeManager* dédiés au pilotage d'une plate-forme d'exécution bas-niveau.

Les adaptations sont également contraintes par les capacités de ces *RuntimeManager*. Pour la gestion de ces adaptations une architecture en couches est proposée, d'un coté la couche système est composée de serveurs équipés de *RuntimeManager* de l'autre une couche de gestion d'architecture est composée d'un gestionnaire d'adaptation qui construit et interprète un modèle d'architecture. L'architecture distribuée proposée centralise donc l'adaptation sur une couche qui prend les décisions de modifications d'architecture. Rainbow n'est pas à proprement parler une implantation de Model@Runtime au sens où l'abstraction qu'il propose n'est pas exploitable directement par les composants et n'a d'existence que dans le gestionnaire d'adaptation qui assemble toutes les représentations pour en faire un modèle d'architecture. Par contre on retrouve dans cette approche la notion de modèle déconnecté qui sert à calculer les adaptations sans impacter directement la plate-forme puisque l'exécution est déléguée à un processus dédié. La figure 3.2 illustre le mécanisme d'extraction du modèle d'architecture et de traitement avant l'application sur les serveurs de l'application dynamique distribuée dirigée par une approche Rainbow.

FIGURE 3.2 – Architecture du projet Rainbow



Rainbow offre donc une solution pour la gestion du cycle de vie d'une application

distribuée tout en prenant en compte le problème d'hétérogénéité des noeuds d'exécution à la fois dans la représentation mais également dans l'application des adaptations au travers des différentes implantations des *RuntimeManager*. Le modèle de développement préconisé définit une séparation entre composant et connecteur telle que proposée dans la classification de Medvedovic [MT00] mais ne propose pas de modèle de concurrence laissant là encore ce travail à la charge du développeur, à la fois de la plate-forme mais également des composants à déployer. L'implantation de la connexion de composants distribués se fait alors en réutilisant des bus de message et de service tels que CORBA dont l'hétérogénéité n'est cependant pas représentée dans le modèle. En laissant ce modèle de communication à la charge du développeur de plate-forme et de composant, Rainbow s'expose donc à des incohérences lors de l'interconnexion de noeuds différents et ne répond pas à la modélisation de la diversité de communication exprimée dans les besoins du cas sapeur-pompiers.

Le modèle Rainbow est donc désynchronisable des plates-formes, répondant alors aux besoins des DDAS vis-à-vis de l'usage de l'architecture comme état réflexif du système en vue de prendre la décision d'une reconfiguration. D'ailleurs principalement cette approche a été utilisée pour faire de l'auto-réparation de système [GCS03],[CGS+02] qui nécessite alors d'avoir la connaissance globale du DDAS pour prendre une décision d'adaptation. Ainsi il est possible de produire plusieurs modèles et de les valider avant leur déploiement effectif.

Par contre le modèle Rainbow ne prévoit pas de délégation quand à l'interprétation du nouveau modèle d'architecture. Dans une première version ceci est fait de manière centralisée par un interpréteur qui exploite les *RuntimeManager* comme des pilotes de plates-formes en les appelant directement. Plus récemment un modèle de délégation entre ces noeuds centraux a été imaginé pour éviter d'avoir un point de défaillance unique [CHG+04].

Cependant dans cette approche les noeuds ne peuvent eux-mêmes appliquer les modifications et encore moins les propager à leurs voisins. Il est donc impossible dans le modèle de représenter plusieurs sémantiques de synchronisation et encore moins de faire de la communication entre noeuds puisque tout est centralisé. Ce manque ne permet pas de faire de la propagation épidémique de configuration tel qu'il est préconisé dans les réseaux mobiles pour résister à des connexions sporadiques.

En résumé pour la construction des DDAS, Rainbow apporte la vision synchrone/-desynchrone du modèle de réflexion qui est également le concept du Model@Runtime. De plus, les multiples agents d'adaptation que sont les *RuntimeManager* permettent bien de prendre en compte l'hétérogénéité de noeuds. Cependant il manque un support pour l'hétérogénéité des capacités de synchronisation et de communication qui peut exister dans des cas mobiles tels que celui du cas de motivation de cette thèse.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	+	±	+	±	±	+	⊗

3.3.1.10 Darwin

Le modèle de composant Darwin a été présenté par Georgiadis *et al* dans le cadre d'un article s'intéressant aux systèmes auto-organisés [GMK02]. Dans ce modèle les auteurs définissent une désormais classique approche à composant à l'aide de ports mais surtout promeuvent l'utilisation d'une séparation type instance pour les éléments de leurs modèles. Ainsi il est possible de créer plusieurs composants, instanciés de façons multiples et rendant des services (port offert) équivalents définis dans un type. Le modèle est typé, ce qui permet de vérifier que les interconnections entre ports permettent d'assurer la correspondance des types.

Le modèle a été conçu pour gérer également la construction de l'architecture globale avec une notion de contraintes, dont l'inspiration vient du langage Alloy [Jac02]. Ainsi pour une catégorie de composant spécifique ayant exactement un port d'entrée et de sortie, il est possible de construire une chaîne de traitement par assemblage décrit en langage de contrainte et de façon automatique en fonction des composants types disponibles.

Cette programmation par contrainte correspond à la vision opportuniste des interconnections globales que préconisent les auteurs. En effet dans leurs visions les composants doivent s'interconnecter à des "voisins" leurs fournissant un service requis sur simple décision du noeud local hébergeant. Ainsi à la place de faire une modélisation totale du système pour faire une reconfiguration, chaque noeud peut modifier la configuration qui lui est demandée pour rapidement répondre au besoin de résolution de service (au travers des ports).

Pour réaliser la synchronisation d'une telle configuration, chaque instance de plate-forme Darwin embarque une vision de l'architecture globale. Cette vue du système est synchronisée avec la plate-forme Java sous-jacente grâce à un processus dédié nommé *Policy manager*. Darwin exploite pour cela directement un mécanisme de chargement de classe Java pour faire le chargement à chaud des composants lorsqu'un déploiement est nécessaire.

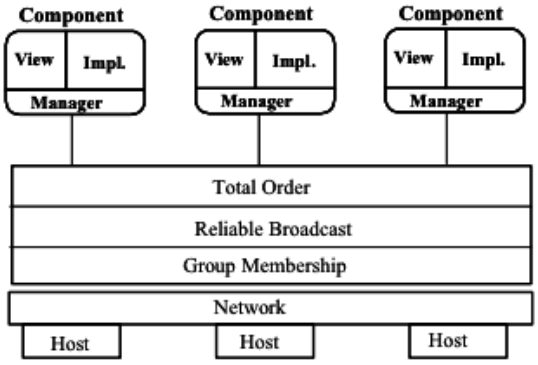
Les interconnexions entre composants sont de type exogène et donc techniquement fournis par le conteneur de composants sous la forme de liaison RPC RMI. Darwin ne peut donc pas proposer de modèle de communication ni de sémantique de concurrence hétérogène.

La synchronisation des noeuds et surtout de leurs modèles de configuration est réalisée à l'aide du projet JavaGroups qui implante un protocole étendu de broadcast réseau disséminant ainsi les modifications. Là encore cette synchronisation est exogène, proposée par la plate-forme et ne permet pas de méthode de synchronisation alternative. De plus la vue d'architecture partagée par tous les noeuds n'est pas désynchronisable, ainsi chaque modification fait l'objet d'un blocage généralisé des noeuds pour appliquer la modification. La vue ne peut donc pas servir de modèle exploratoire car elle impacte directement les plates-formes.

A la manière du Model@Runtime, le projet Darwin propose donc une vision abstraite de l'architecture partagée par l'ensemble des plates-formes pour la réalisation des adaptations. L'idée principale de Darwin est la vision décentralisée de la configuration

pour les systèmes auto-organisés et adaptatifs. A l'inverse d'avoir un noeud décidant seul de la configuration, chacun peut prendre des décisions locales afin de contribuer à l'élaboration d'une configuration globale, trop complexe à calculer en un point. Ce coté opportuniste de la configuration est particulièrement intéressant pour la construction des DDAS de cette thèse, cependant il fait face de fait à un nouveau problème : la synchronisation nécessaire pour faire converger le DDAS vers une configuration commune. Le mécanisme de multicast proposé dans Darwin peut s'exploiter au-dessus d'une couche assurant un ordre total pour gérer les reconfigurations concurrentes et illustré par la figure 3.3.

FIGURE 3.3 – Architecture du projet Darwin



Darwin explicite ainsi une notion de groupe de communication nécessaire pour éviter les conflits d'envoi. Mais ceci ne permet pas la divergence nécessaire des modèles imposés sur les réseaux mobiles non fiables qui nécessitent d'autres modes de propagation dont il faut représenter la diversité, ce qui n'est pas possible dans le modèle Darwin. De même dans les DDAS hétérogènes qui peuvent contenir différentes contraintes entre différents noeuds, il est nécessaire d'exprimer plusieurs notions de groupes de noeuds.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	+	⊗	+	+	⊗	⊗	±

3.3.1.11 Modèle DIF et plateforme Prism-MW

Récemment, en tout début d'année 2012, Malek, Medvidovic *et al* ont publié un article [MMMR12] pour présenter l'approche DIF pour *Deployment Improvement Framework*. Définie sur les bases d'un cas d'étude très proche du cas sapeur-pompiers de cette thèse, cette approche vise à fournir un environnement pour se focaliser sur l'amélioration continue de la qualité de service d'une architecture distribuée. Proche d'une approche Model@Runtime, Malek *et al* définissent une couche de modélisation pour abstraire la notion de représentation abstraite servant à la fois à l'intercession et la réflexion. Ce modèle est donc dédié à un usage en continu et vise à faciliter l'évaluation et le redéploiement du système lui même représenté dans le modèle DIF. L'idée principale de l'approche est d'avoir un modèle désynchronisé des plates-formes servant

d'état initial pour des algorithmes d'optimisation, et qui vise à fournir une nouvelle architecture optimisée de cet état initial. Pour cela ce modèle décrit les noeuds physiques de déploiement mais également la notion de composants ainsi que les propriétés de QoS qui les accompagnent. Les relations entre composants sont simplifiées pour obtenir uniquement l'information de QoS qui peut en être déduit. Plusieurs algorithmes d'exploration sont également évalués pour répondre à l'optimisation multi-axiales des critères de QoS qui est quasiment exponentielle. Un approche utilisant une exploration par un algorithme Greedy [CLRS01] et une autre utilisant une approche génétique sont alors évaluées [Gol89], [Whi94]. Un ensemble d'outils intégré dans Eclipse comme l'environnement DeSI permet de suivre l'impact des différents algorithmes sur l'optimisation d'un modèle DIF. Le lien avec les plates-formes de déploiement est laissé libre et est non spécifié dans le modèle, un exemple est donnée avec une implantation capable de piloter un middleware nommé Prism-MW. Si le modèle abstrait DIF simplifie les notions de relations et communication entre composants le middleware Prism implante lui les notions concrètes de connecteurs et composants

Le lien avec le modèle de développement et le modèle de déploiement n'est donc pas explicite dans cette approche ne permettant pas ici d'effectuer de développement continu nécessaire pour les DDAS. Ce modèle ne permet pas de modéliser l'hétérogénéité des noeuds ni des stratégies de synchronisation des modèles DIF, rendant l'approche exploitable uniquement sur des grilles homogènes et centralisées. En effet le manque de synchronisation explicite des modèles DIF ainsi que le lien difficile avec le modèle de déploiement oblique à déploiement l'algorithme d'optimisation sur un seul noeud central.

Cependant le middleware Prism-MW sous-jacent à cette approche définit lui une notion de composants et de connecteurs qui se retrouvent alignés sur les besoins du modèle de développement idéal d'un DAS. Il définit de plus une notion d'ordonnanceur de port qui permet également de construire le modèle de concurrence nécessaire, même si ce dernier n'est pas inclus dans le modèle de développement. En résumé l'approche de Malek, Medvidovicse rapproche du Model@Runtime et permet de connaitre les critères d'application de plusieurs optimisations multi-axiales. Le middleware Prism-MW pose lui des briques intéressantes pour la construction d'un modèle à composant pour un DAS mais nécessite la définition de la distribution (noeud de calcul dans le modèle). Si cette approche ne permet pas la construction d'un DDAS en l'état, elle réutilise et pose un certain nombre de concepts très proches qui pourraient par exemple servir de cible de déploiement pour l'approche proposée dans cette thèse. L'approche multi-axiale pour l'optimisation est elle rediscutée dans la section validation de cette thèse.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
±	+	+	⊗	±	±	⊗	⊗

3.3.2 Agent et Acteurs

Comme Guerraoui le rappelle dans [Gue99] à propos de la définition d'une bonne abstraction pour réaliser un système distribué, tout est question de la granularité avec laquelle ont veut manipuler le système. Ainsi pour la construction d'un DDAS toute

la question est de savoir quel est le bon niveau pour définir une pièce logiciel du système. D'un coté les composants se focalisent sur le typage des fonctionnalités et des dépendances via un contrat de ports tandis que par opposition les agents logiciels se focalisent sur les échanges et traitements de données métiers [Jen00]. Ainsi on peut définir les agents comme des entités autonomes [FG97] n'échangeant que des messages entre elles, sans pour autant fournir de notion de tâche appelable derrière un port par exemple. De manière très schématique, ce paradigme peut être vu comme l'association d'un paradigme objet et d'un modèle de concurrence et de communication basé sur des échanges de messages uniquement, on parle alors d'objet actif. En assemblant plusieurs de ses agents on peut alors les faire collaborer pour par exemple simuler des systèmes d'intelligence distribués.

La programmation par agent s'est donc focalisée sur la distribution du logiciel et non sa réutilisation donnant lieu à un modèle d'échange et de concurrence par message précis. De plus ce paradigme a amené des travaux autour de l'auto-réparation et des modèles de supervision exploitables sur de multiples noeuds de calcul. A la fois opposées et complémentaires des approches par composants, les approches par agents définissent de nombreuses solutions pour le développement de système distribué. Cette section détaille les projets qui rentre dans ce paradigme de programmation tout en extrayant les éléments qui sont à retenir dans une solution pour les DDAS.

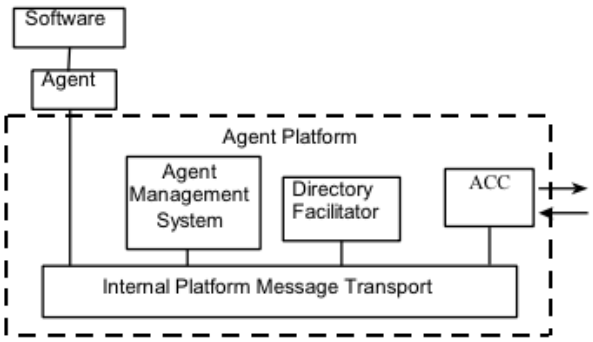
3.3.2.1 Jade

Jade [BPR99] est le nom d'un framework dédié à la construction de système complexe et distribué à l'aide du paradigme *multi-Agent*. Contenant à la fois un modèle de développement et un modèle de déploiement, Jade définit également un middleware assurant la communication par message entre agents. Ce projet est également une implantation de la norme FIPA [15] [Fip97] qui définit le cadre d'un framework en normalisant par exemple la façon de communiquer entre agents (Agent Communication Channel), entre un administrateur et la plate-forme (protocole AMS), etc... Chacun de ces rôles est pris en compte par un agent particulier de la plate-forme qui respecte ces protocoles de communication, l'architecture de ceux-ci est illustrée avec la figure 3.4.

JADE pour Java Agent DEvelopment Framework est donc dédié à l'implantation d'agents pour la machine virtuelle Java. Le modèle de concurrence adopté pour les échanges est unifié, tous les échanges se font par des échanges de messages et chaque Agent dispose de son propre processus implanté sous la forme d'un thread en Java. Cette communication par message peut être réalisée entre les agents d'une même machine virtuelle mais également à distance via un middleware dédié et exogène (hébergé par la plate-forme) implanté en Java sous la forme d'échange RMI. Malgré l'utilisation de RMI le bus de transport de message est asynchrone vis-à-vis des agents et possède son propre thread dédié. Un pont implantant la norme OMG IIOP [GS99] permet de faire le lien entre une plate-forme JADE et un autre framework à Agent. Un agent n'est pas seulement une entité réagissant au stimuli extérieur mais permet aussi de façon autonome de déclencher un traitement. Dans son modèle de concurrence JADE permet

15. http ://www.fipa.org/

FIGURE 3.4 – Modèle de référence de l'architecture FIPA



de définir ce type de modèle comportemental et ainsi permet de définir des tâches périodiques, etc...

L'implantation de JADE et de ce fait le modèle de concurrence pour Java qu'il propose a été évalué vis-à-vis de développeurs différents. Il ressort que pour la construction de système distribué et notamment pour tous les cas de raisonnements distribués, le retour est positif quant à la qualité de l'abstraction. Par contre le modèle de développement de JADE ne permet de proposer des méthodes de communication alternative, telles qu'une propagation gossip au lieu et place de l'implantation RMI. Ainsi la vision endogène de la plate-forme de communication limite les cas d'usage à des communications fiables. Par contre le framework à Agent permet lui de prendre en compte beaucoup de dynamicité quant à l'apparition et disparition des agents. Enfin la standardisation de l'agent d'administration permet de faire des mises à jour à distance de la plate-forme mais aucun mécanisme n'est prévu pour effectuer la synchronisation de ces dernières. Le mécanisme d'administration correspond à l'usage d'un administrateur centralisé tel que Jasmine [16] pour les plates-formes Fractal.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
+	±	±	+	⊗	⊗	⊗	⊗

3.3.2.2 S4

La plate-forme nommée S4 [NRNK10] est dédiée à la création de plate-forme permettant de lourds calculs distribués, tels que ceux nécessaires pour les moteurs d'indexations et de recherche ou encore pour les cas d'usage de fouille de données. Principalement développé par Yahoo ! Labs, S4 pour Simple Scalable Streaming Machine est une plate-forme à Agent dédiée aux traitements de flux de données non bornés.

Ce type de projet se place comme un framework d'architecture grain fin par opposition aux composants qui sont alors vus comme une abstraction gros grain. En d'autres termes, ce framework permet de finement définir la sémantique du système de traitement de message par le biais d'association d'objets actifs et de traitement sur les messages

16. jasmine.ow2.org

entrants.

Le modèle de programmation proposé est similaire à celui de MapReduce [DG08]. Typiquement ce modèle permet de découper des tâches en sous-tâches que l'on peut exécuter en parallèle (Map) puis attendre et collecter les résultats (Reduce). De façon très schématique le projet S4 réalise ce principe au-dessus d'une architecture à Agent, en définissant que les traitements sont hébergés pas des agents capables alors de prendre en compte des fragments de messages à traiter.

Ces messages qui sont en fait des unités de traitement sont nommés *Processing Elements (PE)* et sont alors dispatchés sur des *Processing Node (PN)* qui sont alors réalisés à l'aide d'Agents. Chaque instance de PE est alors composée de quatre éléments : une classe de fonctionnalité, le type d'évènement consommé par ce PE, un nom de clé qui définit la valeur portée, une valeur portée et non mutable. Chaque changement de valeur donne lieu à la création d'un message. Chaque PE est par la suite envoyé à un PN grâce à un routage basé sur une fonction de hashage du nom de clé.

S4 se base sur un modèle de topologie centralisé grâce au projet ZooKeeper [17] [HKJR10], la communication de type TCP/IP entre noeuds est endogène et fixé par la plate-forme qui contrôle la fréquence et le routage des messages. S4 n'est pas à proprement parlé un projet pour la création de DDAS, cependant le modèle de concurrence au niveau programmation permet de faire le lien entre le modèle architectural d'exécution basé sur des noeuds et un modèle orienté donnée. Les systèmes de traitement et d'indexation peuvent tout à fait rentrer dans la catégorie de systèmes DDAS et ceci d'autant plus avec l'arrivée du modèle économique cloud computing qui impose un usage élastique des noeuds de calculs. Ainsi le modèle de DDAS proposé dans cette thèse s'inspire de ces notions de messages types et de modèle de concurrence pour pemettre le développement de DDAS de traitement de données non plus au dessus d'agent mais de composant ayant un cycle de vie connu.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
+	±	±	⊗	⊗	⊗	⊗	⊗

3.3.3 Model@Runtime

Le paradigme *Model@Runtime* a émergé dans la communauté de la modélisation logiciel il y a environ 5 à 6 ans sous la forme d'un premier workshop colocalisé à la conférence MODELS'06 [BBF07] suite à un papier de Zhang, Cheng *et al* à la conférence ICSE'06. Dans sa première version ce paradigme s'est attaqué à la construction de logiciels dynamiquement adaptables dont la complexité dépassait le seuil manipulable par les méthodes de constructions logicielles objets classiques. L'idée était de garder la couche de modélisation qui résolvait les problèmes de construction logicielle complexe au moment du design et de l'inclure directement dans la plate-forme. Ainsi présent également au moment du "runtime", ce modèle pouvait alors servir l'ensemble des outils qui manipulent le système pendant son fonctionnement afin d'en faire un système dynamiquement adaptable. D'un certain point de vue ce modèle promu au rang de super ADL

17. http ://zookeeper.apache.org/

permet de représenter le système mais permet surtout d'interagir avec lui. En effet l'idée principale de cette approche est de pouvoir synchroniser ce modèle avec la plate-forme pour l'impacter concrètement. Le modèle devient donc une représentation qui se veut indépendante de la plate-forme et permet d'interagir de manière plus simple et plus sûre avec elle. Plus simple, car le modèle est construit pour être une abstraction plus proche de l'utilisateur (le problème qui calcule les évolutions) et plus sûr car toutes les étapes de modifications peuvent être faites avant la synchronisation avec la plate-forme évitant alors le coté irréversible de la modification.

Modèle (de ou au) Runtime Dans sa première version le Model@Runtime a servi à représenter le comportement d'un système, en l'occurrence un automate d'exécution d'écrit en PetriNet. Dans ce cas d'usage le modèle est la plate-forme au sens ou le même PetriNet est directement interprété par la plate-forme pour en déterminer son comportement, en l'occurrence un certain nombre de transitions tirées et exécutées à la réception d'un message. Puis dans un article suivant, Blair, Bencomo, R.France *et al* définissent le Model@Runtime[BBF09b] comme une abstraction devant piloter les systèmes adaptatifs et non être le système lui-même.

Modèle structurel ou comportemental En effet l'abstraction doit être plus simple et moins coûteuse que la réalité (briques exécutables) et non être la réalité elle-même pour offrir une couche simplificatrice sans cela le modèle s'impose des contraintes d'expressivité dues à sa cible d'exécution. De cette divergence va également émerger la différence entre les modèles comportementaux, le plus souvent représentés sous forme d'automates et les modèles d'architecture structurels qui définissent des interactions plus proches du modèle MAPE. L'approche par modèle structurel s'est assez naturellement rapprochée des paradigmes composants [MBJ+09a] afin de définir la granularité de briques logiciel manipulables.

3.3.3.1 Genie

L'approche proposée par Bencomo *et al* nommée Genie [BB06],[BGF+08],[BSBG08] propose d'exploiter un ensemble de DSL pour représenter chaque axe de la construction d'un système. Exploités directement dans la plate-forme ces DSL permettent d'interagir avec elle avec un haut niveau d'abstraction. Chaque DSL prend en charge une problématique qui permet de construire et d'assembler un ensemble de composants issus du modèle OpenCOM développé à l'université de Lancaster. On trouve alors un DSL pour la définition des composants, un autre pour l'expression de la variabilité du système [BSBG08] inspiré des lignes de produits dynamiques, etc... L'originalité de l'approche réside notamment dans sa définition de l'évolution, en effet un DSL est dédié à l'expression de migration d'état du système. Celui-ci permet de modéliser l'ensemble des mises à jour correctives qui peuvent intervenir dans le cycle de vie du DAS par transition d'état.

Que ce soit en exploitant un DSL de ligne de produit dynamique pour exprimer l'ensemble des variations d'un système ou en exprimant l'ensemble des variations d'état,

l'approche proposée dans Genie se limite uniquement à des mises à jour prévues au moment du design. Il est donc de fait impossible de reconfigurer le système dans un état non prévu au design et encore moins d'appliquer le principe du développement continu pour mettre à jour les composants. De plus le modèle OpenCOM ne permet pas de représenter la notion de noeud et encore moins leur hétérogénéité et doit donc être piloté par un programme extérieur pour ce type de gestion et le déploiement.

Néanmoins vis-à-vis des besoins pour la construction d'un DDAS l'approche Model@Runtime proposée par Genie apporte la notion de modèle désynchronisable dont le processus de construction exploité au design est exploitable au runtime pour calculer les évolutions, soit en redérivant une nouvelle configuration grâce à l'approche ligne de produit, soit en appliquant une transition précalculée. L'approche Genie exploite également un modèle structurel de composant synchronisé avec le runtime et non directement interprété ce qui lui laisse la liberté d'avoir une abstraction plus riche que la plate-forme.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	⊗	⊗	+	±	+	⊗	⊗

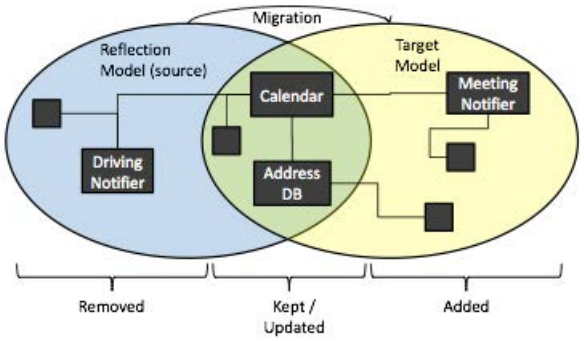
3.3.3.2 SmartAdapters / Modèle ART

Le projet SmartAdapters a été développé dans le cadre du projet européen DiVA [18] et dans le cadre de la thèse de Brice Morin [MBJ+09b],[Mor10a]. La vision de ce projet est d'exploiter une couche de modélisation pour piloter une plate-forme à composants. Pour cela un lien causal bi-directionel est établi entre une représentation modèle et la plate-forme. Ainsi définissant une approche Model@Runtime, les auteurs proposent donc de faire remonter les évènements de la plate-forme directement dans le modèle et à l'inverse d'impacter toute modification du modèle sur la plate-forme. Le modèle devient alors une couche de réflexion du système mais également d'intercession puisque qu'elle permet l'interaction avec lui. Un seul modèle est alors sauvé dans tous les cas, pour représenter le système sous étude. Pour appliquer les modifications faites au niveau modèle, l'approche repose sur un opérateur de comparaison qui permet de détecter si les éléments structurels du modèle (principalement des composants) sont ajoutés ou supprimés. La figure 3.5 représente cet opérateur qui considère deux modèles, l'un est le modèle courant représentant la plate-forme, l'autre celui cible que la plate-forme doit atteindre.

L'approche SmartAdapter repose sur un modèle de composant nommé ART pour (at runtime) qui définit des notions de composants, ports et service fortement alignés avec le standard SCA. Chaque composant ART se définit donc à l'aide d'un fragment de modèle qui définit sa structure. Pour l'assemblage de ces composants l'approche propose de reprendre les outils de modélisation de ligne de produit dynamique et de définir chaque facette qui composant un DAS pour un fragment de modèle associé à un variant dans un feature modèle. Ainsi à l'aide d'un tisseur d'aspect de niveau modèle est possible de composer la construction du modèle final qui respecte l'ensemble des variants sélectionnés. Proche du projet Genie sur ce point l'approche SmartAdapter

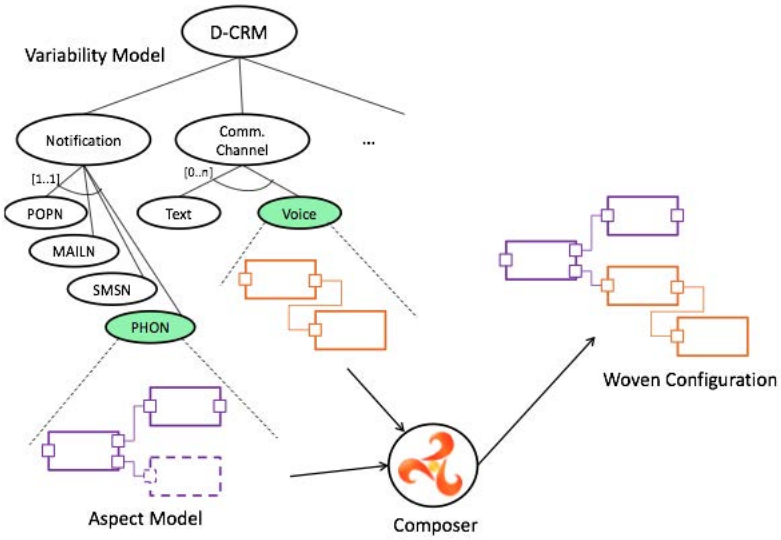
18. http ://www.ict-diva.eu/

FIGURE 3.5 – Opérateur de comparaison de modèle de SmartAdapters



représente donc les variations d'un DAS sous la forme de variants, dont la sélection peut changer pendant le fonctionnement et donner lieu à un nouveau tissage et donc un nouveau modèle devant être synchronisé avec la plate-forme.

FIGURE 3.6 – Opérateur de tissage d'aspect SmartAdapters



Pour le déploiement effectif des composants l'approche a d'abord été développée au-dessus de la plate-forme Fractal puis au-dessus de la plate-forme OSGi afin d'obtenir la capacité de chargement à chaud. Cependant le modèle ART se base sur une approche à service et n'encapsule donc pas dans son modèle de représentation d'autres patrons d'échange tels que les évènements. De plus il n'est pas possible de représenter la notion de noeud ni la topologie du réseau et encore moins la capacité de synchronisation. En résumé l'approche proposée dans le projet SmartAdapter peut se séparer en deux parties, d'un coté la composition de modèle d'architecture par tissage de fragments de modèles, de l'autre une approche Model@Runtime pour les DAS reposant sur un mo-

dèle de composant nommé ART. Ce deuxième aspect est une brique qui répond à de nombreux points nécessaires pour la construction du cas sapeur-pompiers de motivations, la capacité de synchronisation et désynchronisation est essentielle pour permettre d'exploiter plusieurs méthodes de calculs de modèles. Le modèle ART souffre également d'un manque quand au lien avec le modèle de développement , sur la notion de concurrence de ses ports, et sur la séparation claire entre composant et connecteur. Défini pour l'adaptation de DAS, ART ne permet pas, tel quel, la construction de DDAS, il est nécessaire d'y ajouter la notion de distribution et d'hétérogénéité des capacités de communication, mais cette thèse s'inspire de ces travaux.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	⊗	⊗	+	±	+	⊗	⊗

3.4 Dissémination et distribution des adaptations

La boucle d'adaptation MAPE imaginée au début des systèmes adaptatifs est maintenant confrontée au problème de la distribution. En effet, dans le cas où la distribution ne concerne plus uniquement le système et ses composants mais également le calcul des états des noeuds et donc l'adaptation, arrive irrémédiablement le problème de coordination. L'introduction de multiples points de décision revient alors à l'introduction de multiples boucles MAPE qu'il faut alors synchroniser en envoyant les nouveaux et coordonner pour éviter les choix concurrents. Au croisement de l'algorithmique distribué et des systèmes adaptatifs, ce problème aborde plusieurs aspects, dont le problème de dissémination efficace et le problème de consensus ou convergences dans les protocoles épidémiques tels que Gossip. Des nouveaux patrons de conception, comme ceux de Wolf *et al* [DWH07] et Weyns *et al* [WSG+12] ont d'ailleurs été proposés pour la conception de ses architectures "émergentes", c'est à dire issues de plusieurs décideurs de configuration. Cette sous-section traite des différents projets permettant soit la dissémination de reconfiguration de façon opportuniste (détaillée en sous-section 3.4.2), soit permettant la construction de configuration par agrégation et collaboration de noeuds participants (détaillée en sous-section 3.4.1).

3.4.1 Disséminiation incrémentale de modèle d'adaptation

Un des premiers patrons de coordinations d'adaptation mis en avant récemment par Weyns *et al* [WSG+12] est : coordination du contrôle. Celui-ci est défini pour les cas ou l'élection de leader n'est pas faisable pour des besoins métiers, et à la place cette coordination vise à faire émerger la solution efficace des participants de manière collaborative. Dans ce cas de figure, chaque boucle MAPE communique avec les autres en pair-à-pair afin de valider ou non chaque adaptation.

FlashMob, modèle Koala

L'algorithme FlashMob est issu des travaux de thèse de Daniel Sykes [SMK11],[Syk10]. L'objectif de ce projet est de proposer une solution totalement décentralisée pour la

construction d'architecture de système adaptatif, à l'inverse donc des solutions à hyperviseur telles que celles proposées dans Rainbow et Acme. En postulat, Sykes élimine toute élection d'un leader ou d'un point central, le but est à l'inverse de faire émerger la configuration des besoins des différents noeuds qui participent alors activement à l'élaboration de la solution.

Le modèle proposé par FlashMob définit 3 couches : la première définit le but et les besoins de l'architecture, la deuxième les migrations, tandis que la troisième définit le plus bas niveau à savoir la plate-forme et les composants qui y sont hébergés. La couche composant est définie à l'aide de composants définis grâce au modèle Koala [VOVDLKM00]. Le principe de base du protocole est à l'image des protocoles Gossip de type agrégation. Pour calculer une nouvelle configuration, chaque noeud peut envoyer à l'aide d'un protocole Gossip, un fragment de configuration aux autres noeuds. Ce fragment est une modification incrémentale de l'état courant et ne contient que les éléments modifiés, soit une nouvelle relation entre composants, soit l'installation d'un nouveau composant. Chaque noeud peut ensuite accepter ou non la reconfiguration et même la modifier tout en la propageant afin de faire émerger une solution qui satisfasse l'ensemble des noeuds. Une fois acceptée par tous , la reconfiguration est appliquée. Chaque noeud peut ainsi participer à l'élaboration et l'amélioration du modèle d'architecture globale. Tout comme les approches Model@Runtime, FlashMob nécessite donc une modélisation même partielle de l'architecture désynchronisable de la plate-forme afin de pouvoir l'échanger de manière opportuniste à l'aide d'un Gossip. La divergence est également une idée qui se dégage de cette solution. En effet ce type d'approche laisse diverger les noeuds du DDAS, au sens où chacun ne connaît qu'une approximation de l'état de ses voisins, ce qui leur permet de faire émerger une solution sans imposer un consensus obligatoire entre tous les noeuds ce qui serait inefficace sur un large réseau.

En revanche l'approche de Sykes ne définit pas comment le protocole peut résister à la perte de noeud pour décider ou non de la validité de la solution émergente. De même la notion d'état n'est pas clairement définie, l'approche repose sur une définition des possibilités d'hébergement des composants sur les noeuds sans pour autant permettre leurs mises à jour, introduisant du même coup une assomption de monde clos. Le problème est à peu près le même pour le type de topologie que l'approche peut exploiter. Enfin l'assomption principale de convergence de Skypes est que tous les noeuds prennent les mêmes décisions d'acceptation ou non vis-à-vis des propositions. Implicitement ceci impose d'avoir le même type de raisonnement sur chaque noeud et ne permet pas de gérer les conflits en cas de décision dans le même temps.

L'approche de Skypes permet d'imaginer des DDAS capables de faire émerger les configurations de chaque noeud du réseau et permettant ainsi de gérer réellement des systèmes larges échelles, dont la reconfiguration globale est complexe. La notion d'état est cependant problématique, pour la gestion de l'hétérogénéité des adaptations d'une part, et pour gérer le design continu et éviter l'assomption du monde clos des types de composants, ou de la topologie réseau, d'autre part.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
⊗	±	⊗	⊗	⊗	+	⊗	⊗

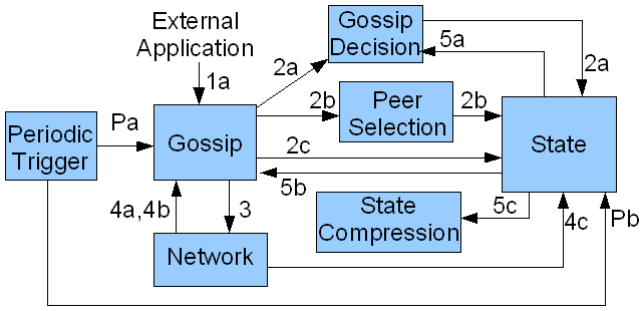
3.4.2 Algorithmes de dissémination dans des réseaux complexes

Weyns *et al* définissent dans leur classification un patron de conception : partage d'information. En d'autres termes, la boucle MAPE partage les informations nécessaires de proche en proche pour assurer la cohérence de l'exécution sur chaque noeud. Mis en corrélation avec les besoins des disséminations sur les réseaux mobiles, ceci induit d'avoir des algorithmes de disséminations d'état pouvant résister à différentes topologies réseaux, et cette sous-section traite des projets de cette catégorie.

GossipKit

Le projet GossipKit [Tai10] est dédié à la construction générique et réutilisable de protocoles de type Gossip. Ce framework [LTB+07],[LTB08] permet la définition de modules réutilisables qui par assemblage permettent de construire un middleware implantant une propagation Gossip avec des propriétés spécifiques pour un cas d'usage. L'assemblage des modules se fait suivant une architecture à gros grain commune à tous les protocoles Gossip illustrée dans la figure 3.7.

FIGURE 3.7 – Architecture commune des protocoles Gossip : GossipKit



Chaque module implantant un rôle dans cette architecture gros grain peut se composer lui-même à l'aide de micro-modules plus réutilisables (architecture grain fin). La composition des modules et micromodules se fait à l'aide d'un modèle évènement et permet de construire des approches pro-actives (pooling) ou réactives. Les architectures GossipKit se génèrent à l'aide d'une configuration manuelle basée sur XML ou dérivée à partir d'un DSL dédié. Le middleware résultant exploite une version Java du projet OpenCOM et support des reconfigurations dynamiques pour changer le protocole Gossip déployé à chaud.

La vocation de GossipKit n'est pas de fournir un support pour la modélisation généraliste d'un système distribué par exemple à base de composant. A l'inverse il se focalise sur la modélisation d'un mécanisme de propagation d'état (donnée) qui exploite un paradigme d'architecture composant pour fournir un service Gossip à une application tiers. Il n'offre pas de support pour la modélisation *offline* d'un système distribué, ni pour l'hétérogénéité de ses noeuds et/ou de leurs déploiements.

Cependant l'usage extensible et reconfigurable à chaud des plates-formes résultantes

de GossipKit apportent de nombreux éléments pour la construction d'un DDAS. Premièrement, malgré la complexité annoncée pour le développeur, les auteurs de GossipKit préconisent un modèle d'échange évènementiel afin d'assurer un couplage lâche entre leurs modules. Ce modèle d'évènement assure la composition de ceux-ci via des boucles d'évènements auxquels ils s'abonnent. Ces abonnements et le modèle asynchrone permettent également les mises à jour à chaud des modules, car ils explicitent le cycle de vie. Ainsi on peut mettre en pause les boucles ou les abonnements pour effectuer les mises à jour des modules.

De plus GossipKit définit grâce à ce modèle d'échange, des mécanismes de composition qui permettent de faire collaborer plusieurs instances de protocoles différents Gossip. Apportant ainsi une réponse à l'hétérogénité des communications, GossipKit peut alors faire travailler sur des instances de protocoles différents mais limités à du Gossip. Enfin, l'api de communication apporte un premier niveau de la boucle MAPE et les services de peer sampling peuvent être vus comme un élément allant vers un modèle offline de réflexion des noeuds et de leurs topologie.

Si GossipKit n'est pas une réponse au besoin du cas sapeur-pompiers, sa modélisation des communications distribuées met en lumière des points essentiels pour la construction d'un DDAS. Le modèle de composition évènement asynchrone est une brique essentielle pour la mise à jour et la composition des différents protocoles hétérogènes collaborants. L'approche par DSL de GossipKit s'avère également très expressive et efficace pour modéliser des protocoles servant soit à la dissémination des données soit à des politiques de reconfiguraiton du DDAS.

ConcurPort	DistModel	ComSep	MapeAPI	MapDeploy	AsynReflect	HeteAdapt	DistAdapt
±	+	±	±	⊗	⊗	⊗	⊗

3.5 Synthèse et enjeux

Le tableau suivant synthétise les différents projets étudiés sans la section état de l'art vis-à-vis des critères recherchés pour la création d'un DDAS.

Projet	CPort	DistModel	ComSep	MapeAPI	Deploy	AReflect	HeteAdapt	DistAdapt
Fractal	⊗	⊗	±	+	⊗	⊗	±	⊗
ArchJAVA	±	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Frascati	±	⊗	+	+	⊗	⊗	±	⊗
OSGi	⊗	±	⊗	+	+	⊗	⊗	⊗
iPOJO	⊗	⊗	⊗	+	+	⊗	⊗	⊗
Sam	⊗	±	⊗	+	+	⊗	±	⊗
SOFA2	⊗	±	±	+	+	⊗	±	⊗
Rainbow	⊗	+	±	+	±	±	+	⊗
DARWIN	⊗	+	⊗	+	+	⊗	⊗	±
PrismPW	±	+	+	⊗	±	±	⊗	⊗
Jade	+	±	±	+	⊗	⊗	⊗	⊗
S4	+	±	±	⊗	⊗	⊗	⊗	⊗
Genie	⊗	⊗	⊗	+	±	+	⊗	⊗
Art	⊗	⊗	⊗	+	±	+	⊗	⊗
FlashMob	⊗	±	⊗	⊗	⊗	+	⊗	⊗
GossipKit	±	+	±	±	⊗	⊗	⊗	⊗

Cette section état de l'art couvre des projets ayant des buts et objectifs très différents mais tous ont en commun l'élaboration d'un système dynamique ou/et distribué. Il ressort de l'étude de l'état de l'art que le design et l'implantation d'un système de type DDAS reste un challenge actuellement. En effet si certains des critères identifiés sont pleinement abordés par une ou plusieurs approches, il subsiste une séparation entre les méthodes de designs, de déploiements et celles de développements. Cette séparation disperse ces propriétés dans des familles de solutions qui doivent être composées afin d'avoir un système à la fois distribué et dynamiquement adaptable.

Ce constat rejoint les conclusions de Medvidovic et plus récemment celles de Crnkovic. Les modèles de design se spécialisent suivant les cas d'usage, se focalisant alors sur tel ou tel objectif. Notamment, si les approches à composants et à services ont prouvé leur capacité à réduire le couplage entre les briques logiciels amenant ainsi la flexibilité nécessaire pour l'adaptation, les approches à Actor/Agent permettent quant à elles d'expliciter et de prendre en compte les problèmes de concurrence inhérents à un usage distribué. Le constat de Crnkovic s'extrapole donc au delà des modèles à composants, par exemple l'approche du projet S4 se spécialise sur les flux de données tandis que Jade sur la communication multi-agents. De manière plus précise, la plupart des modèles de composants hébergent donc des capacités d'adaptation via une API MAPE. Seulement certains étendent cette API au chargement à chaud de code comme les approches dérivées d'OSGi. Les approches à agents définissent elles, pour la plupart, des modèles de communication séparés du code métier (médium de communication) et surtout définissent un modèle de concurrence clair ce qui est rarement intégré dans les modèles de composants.

La spécialisation des approches explique donc cette dispersion mais laisse apparaître également que l'adaptation de noeuds hétérogènes et surtout l'adaptation distribuée sont les deux grands manques identifiés (représentés par les deux dernières colonnes du tableau). Seul SAM, Ra inbow et SOFA2 proposent un support pour la gestion de noeuds hétérogènes et surtout seul DARWIN propose d'intégrer le modèle de synchronisation dans les plates-formes et intègre un modèle de topologie pour ce faire. La gestion de l'hétérogénéité est d'ailleurs le plus souvent effectuée par l'utilisation d'API ou Runtime commun tel que le modèle Fractal qui possède une implantation symétrique en C et Java.‘ Ceci pose problème pour exploiter un raisonnement qui exploite les spécificités de chaque plate-forme, ou plus simplement pour connaître les déploiements de composants compatibles sur telle ou telle plate-forme. La plupart des autres approches délègue la synchronisation et mise à jour des noeuds à des programmes tiers tels que des superviseurs. Ces superviseurs prennent alors la charge de la modélisation topologique et synchronisent de manière autoritaire les noeuds.

Cette centralisation pose problème pour les déploiements sur des environnements pairs-à-pairs, mais au-delà, le fait de ne pas garder le modèle de déploiement dans le système interdit tout déploiement déclenché depuis un noeud. Ceci limite donc la création de système auto-adaptatif. Le manque de lien direct entre le modèle de déploiement et celui de design introduit également des liens implicites entre les plates-formes ce qui pose d'autant plus problème pour calculer les adaptations. Par exemple les liaisons dans le modèle DOSGi sont uniquement connues de la plate-forme et n'exploitent que l'infor-

mation d'un point d'ancrage de service distant, un fonctionnement similaire est présent dans le modèle SCA. Enfin le manque de liaison avec le modèle de développement complexifie le design continu puisque ne permet pas facilement la répétition de la phase de déploiement avec la traçabilité des modifications de design.

La réflexion asynchrone (6 ème colonne) est également une fonctionnalité manquante dans la plupart des approches, hormis les dérivés de Model@Runtime qui abordent spécifiquement ce problème tels que Genie ou Art. Les autres approches expriment leurs adaptations de façon couplée à la couche d'intercession Ce couplage synchrone limite l'usage des récents résultats d'algorithme distribués tels que Gossip qui nécessitent une dissémination asynchrone pour exploiter les réseaux pairs-à-pairs.

En synthèse, il est donc identifié que l'adaptation distribuée reste un challenge pour les approches de construction de DAS actuelle et notamment autour des approches à composants et agents. Cette synchronisation de DAS est d'autant plus limitée de part le manque de découplage entre la couche d'intercession et la couche de raisonnement/dissémination. Enfin la gestion de l'hétérogénéité des noeuds et de leurs capacités d'adaptation reste également un problème, en partie dû au manque de lien entre le modèle de déploiement et le modèle de design et développement.

Deuxième partie

Thèse et Application

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.
Antoine de Saint-Exupéry

Cette partie est dédiée à la contribution de cette thèse qui s'articule autour d'une approche Model@Runtime pour les systèmes distribués. Le chapitre 4 introduit donc les concepts généraux de la solution proposée qui vise à définir une couche de modélisation qui couvre les problèmes identifiés dans l'état de l'art, à savoir la gestion de l'hétérogénéité des plates-formes et leurs capacités de synchronisation. Pour être adaptable dynamiquement, le logiciel fonctionnant sur ces plates-formes doit assurer un certain nombre de propriétés. Pour assurer structurellement celles-ci, le chapitre 5 détaille les éléments retenus issus des modèles à composants ainsi que les extensions nécessaires pour modéliser explicitement les communications distantes. Puis le chapitre 6 détaille plus en profondeur l'approche Model@Runtime proposée, notamment le modèle de noeud, le processus de synchronisation entre un modèle d'architecture et une plate-forme et enfin la notion de groupe permettant la coordination d'adaptations distribuées. Cette partie se conclut par une discussion sur l'impact du modèle proposé sur les développements de systèmes auto-adaptatifs.

Chapitre 4

Concepts généraux de la contribution

4.1 Idée générale

4.1.1 Un modèle commun pour les étapes du cycle de vie logiciel

La contribution de cette thèse est fondée sur l'idée d'utiliser un modèle comme abstraction générique pour relier les outils de développement, design et de déploiement pour l'évolution et la maintenance à chaud des DDAS. De façon très schématique ce modèle nommé Kevoree contient à la fois les informations des fonctionnalités disponibles du système (liste des artefacts développés et leur localisation sur des serveurs de binaires associés) ainsi que le résultat de configuration ou d'assemblage. Cet assemblage modélise les nœuds de calcul du DDAS, leurs connexions mais surtout les fonctionnalités associées et donc à déployer. Plus précisément ce modèle s'appuie sur un modèle de développement et de déploiement à base de composants et connecteurs dont une représentation schématique est donnée par la figure 4.1 à l'aide de la définition 1 de contenance d'un élément dans un autre.

Déf 1. *La relation de contenance d'une entité B dans une entité A, exprimée via l'opérateur "/->"*

```
A
/->B
```

possède les propriétés suivantes, inspirées de la sémantique de contenance du MOF [OMG11] :

- *une entité doit avoir un seul conteneur, B ne peut être contenue que par une unique entité A ;*
- *la contenance cyclique est interdite, B ni aucun de ces objets contenus ne peuvent contenir A ;*
- *la relation de contenance définit une inclusion directe, toute suppression de A entraîne la suppression de B, mais B peut être supprimé indépendamment de A.*

La figure 4.1 représente donc un modèle Kevoree contenant une définition de type (T1) et son binaire, ainsi que deux noeuds instances (A et B) contenant chacun deux composants du type précédemment déclaré.

FIGURE 4.1 – Représentation shématique du modèle Kevoree

```
KevModel
|->Type id=T1,version=1
    |-> Binaire URL=http://...
|->Noeud id=A
    |-> Component id=A1 type=T1.v1
|->Noeud id=B
    |-> Component id=B1 type=T1.v1
```

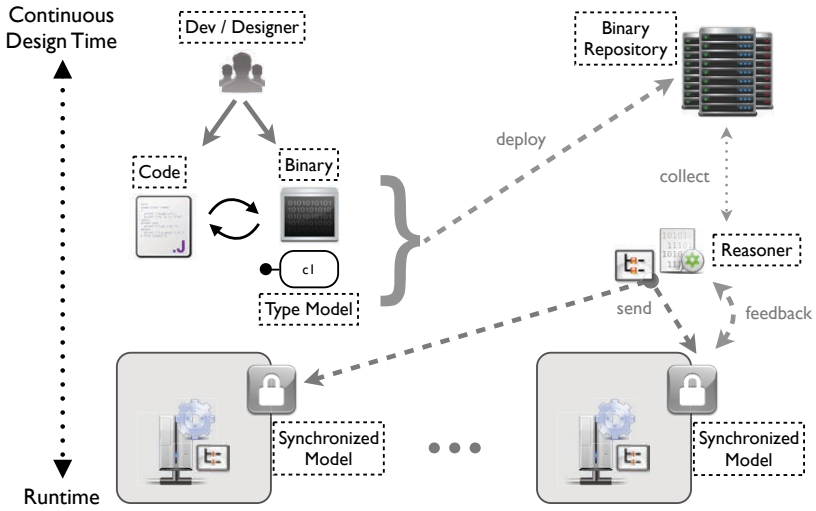
Cet assemblage est construit par diverses méthodes, soit par un humain, soit par un ensemble d'outils et de méthodes que l'on regroupe sous le terme *raisonneur*. Celui-ci englobe toutes les méthodes de composition de configuration, par aspects [MBNJ09a] ou par scripts de reconfiguration (FScript [DL+06],SAFRAN [DGR09]) pour ne citer qu'elles. Le modèle d'instance qui découle de ce design n'est pas statique mais au contraire défini de façon cyclique dans un processus nommé *design continu*. Les fonctionnalités déployées sur un nœud dépendent d'une version binaire compilée des artefacts, une modification (par exemple changement du code et recompilation) de ceux-ci entraîne une mise à jour des instances liées à l'ancienne version. Ainsi par le biais d'une traçabilité des binaires directement dans le modèle et des opérateurs de mutation des instances cette approche de design continu répond au besoin d'évolutivité exprimé dans l'état de l'art.

4.1.2 Mise à jour de DDAS par propagation de modèle

La principale contribution de cette thèse est l'extension du Model@Runtime pour un usage distribué, en définissant d'une part la distribution (les noeuds) dans le modèle et en introduisant les mécanismes pour distribuer le modèle lui-même d'autre part. En effet, Morin *et al* ont défini le processus Model@Runtime pour les systèmes DAS dont la principale caractéristique est la synchronisation à la demande du modèle ce qui permet d'introduire des étapes de vérification avant déploiement. L'objet de la présente contribution est de définir par extension du Model@Runtime un processus permettant de synchroniser plusieurs nœuds et donc de propager une mise à jour de façon asynchrone simplement par dissémination de modèle. Dans ce nouveau processus chaque nœud est responsable de la cohérence de son modèle courant avec sa plate-forme concrète. Chaque modèle reçu déclenche alors localement un processus Model@Runtime qui se traduit par un calcul de différence avec le modèle courant, une mise à jour de la plate-forme en fonction de cette différence et une sauvegarde du nouvel état courant. Le modèle d'architecture existe alors en copie sur chaque nœud, chaque copie contenant les informations des autres nœuds comme l'illustre la figure 4.2. Ainsi pour faire une mise à jour locale ou distante, un raisonneur doit simplement lire l'état courant, modifier le modèle selon différentes approches et le propager aux autres nœuds. L'interaction avec le processus MAPE peut alors se résumer dans cette approche à l'interaction avec le

modèle. En effet l'écoute des modifications de modèle permet de faire le *monitoring* et déclencher l'analyse et planification sur le modèle courant, tandis que l'exécution se résume à la propagation locale et distante d'une version modifiée de ce modèle.

FIGURE 4.2 – Processus Model@Runtime vue d'ensemble



4.1.3 La divergence pour la performance et la résilience

L'état stable d'un DDAS dans cette approche se caractérise par la similarité entre les copies des différents nœuds. Les processus de raisonnement peuvent, quant à eux, être hébergés sur chacun des nœuds, produisant et propageant alors un nouveau modèle lorsqu'une modification est calculée. Dès lors les copies des autres nœuds peuvent diverger soit en raison des délais de la propagation, soit parce que la communication n'a pu être établie, les nœuds n'étant pas joignables. Cette divergence existe de manière inhérente aux communications réseaux dans les DDAS actuels mais est le plus souvent soit restreinte à des cas d'usage centralisés tels que les superviseurs de grille, soit limitée dans le même temps logique dans le cas des clusters (ordre total). Or, l'état de l'art notamment des réseaux pairs-à-pairs montrent une grande diversité des approches qui en profitant de cette capacité de divergence permettent de tenir une forte charge mais surtout une importante résilience aux fautes. Ainsi s'il fallait faire le parallèle avec les bases de données, accepter cette divergence s'apparente à l'acceptation de la perte des propriétés ACID [Pri08] qui a permis de répartir de façon opportuniste les données sans pouvoir en permanence assurer la garantie d'homogénéité à tout moment mais à l'opposé d'assurer une *consistance éventuelle* [Bre00], [GL02], [BGL+06]. A l'inverse donc d'une communication distribuée transactionnelle cette divergence acceptée permet de résister à la perte d'un nœud mais permet également d'en ajouter dynamiquement pour renforcer le système. Ce parallèle est représentatif de la catégorie de systèmes visée par cette contribution : des systèmes distribués dynamiquement adaptatifs qui doivent composer avec des nœuds sporadiques, c'est-à-dire volatils et intermittents. Les systèmes

ultra large échelle [NFG+06] rencontrent également ce problème de divergence de manière inhérente à leur taille et à la fragmentation de leurs développements. Leur travaux arrivent également à la conclusion qu'il faut chercher à maîtriser cette divergence et non la contraindre. La capacité de divergence est assurée dans le modèle proposé en introduisant un cycle MAPE plus un processus Model@Runtime autonome sur chacun des nœuds. Chaque noeud de façon autonome peut exécuter localement des mises à jour et produire un nouvel état modèle qui diverge alors de l'ancienne vision modèle de l'un de ces noeuds voisins. La divergence est donc introduite par l'accès local à l'exécution des adaptations et donc à l'accès en écriture à la copie local du modèle de chaque noeud.

4.1.4 L'hétérogénité des algorithmes de convergence

La dissémination du modèle est la pierre angulaire de cette contribution. En effet les nœuds doivent échanger leurs modèles pour propager leurs décisions de mise à jour et faire converger le DDAS vers un état stable. Dans cette approche, c'est le type de dissémination qui va déterminer les propriétés de cohérence que l'on peut attendre des modèles locaux et donc plus généralement du DDAS. Or là encore, dans l'état de l'art, les algorithmes qui permettent de faire la dissémination d'informations sont aussi multiples que les cas d'usage. De plus les propriétés de cohérence du DDAS peuvent évoluer dans le temps. Par exemple lors d'un changement de topologie ou de connectivité réseaux (ex : passage d'un connexion 3G à du Wifi), ceci se traduit inévitablement par un changement des algorithmes de synchronisation. Pour répondre à cela, Kevoree introduit une entité de première classe spécialement dédiée à modéliser les capacités de synchronisation et donc la sémantique d'échange du modèle entre les nœuds. Cette entité nommée *Groupe* est définie avec les mêmes capacités d'adaptation dynamique que les composants ou les connecteurs. Définie directement dans le modèle comme illustré par la figure 4.3, l'entité groupe exploite donc directement le modèle qu'elle même dissémine pour ses propres mises à jour.

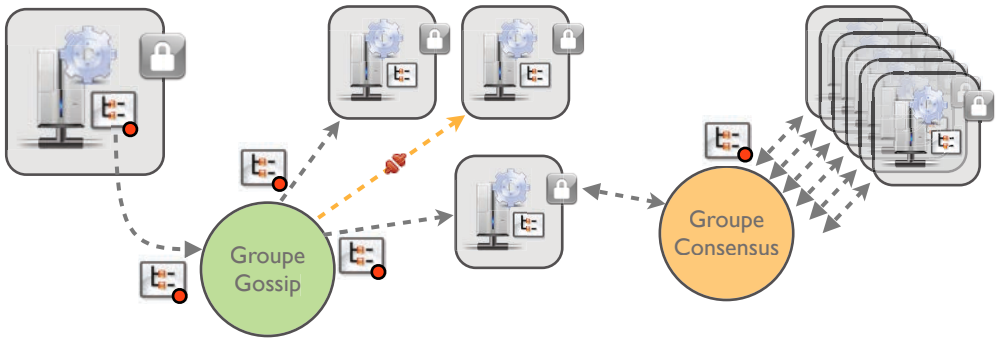
FIGURE 4.3 – Représentation shématique de l'entité *Group* dans le modèle Kevoree

```
KevModel
|->Type id=Gossip,version=1
    |-> Binaire URL=http://...
|->Nœud id=A
|->Nœud id=B
|->Group id=G1 type=Gossip.v1
    |->subNode id=A
    |->subNode id=B
```

Les groupes sont donc à l'écoute des modifications locales et déclenchent la propagation avant ou après l'exécution locale de l'adaptation suivant le type d'algorithme qu'ils encapsulent. Les groupes peuvent être multiples et injectés dans plusieurs localisations du DDAS afin de construire différents groupes de nœuds qui assurent alors différentes propriétés de convergence. Par exemple la figure 4.4 illustre l'usage des deux grandes familles d'algorithmes envisagés. Le côté gauche de cette figure est synchronisé avec un groupe *Gossip* qui réalise un ordre partiel, tandis que le côté droit est synchronisé

en ordre total par un algorithme de consensus assurant la garantie d'acceptation du modèle par une majorité des nœuds de son groupe. Les deux parties sont reliées par un nœud faisant ainsi passerelle entre les deux groupes de nœuds. Ainsi la partie gauche sera plus adaptée sur des nœuds mobiles et résistera à des pertes de connexions (comme illustré par le lien orange) tandis que la partie droite sera adaptée à des nœuds fixes. Mais l'ensemble du système convergera vers la même version des modèles émis.

FIGURE 4.4 – Groupes pour la synchronisaiton des Model@Runtime



Modèle slicing pour gérer la complexité en largeur

Comme annoncé dans les motivations, cette contribution traite également des systèmes à large échelle caractérisés par une complexité en largeur, c'est-à-dire un grand nombre de nœuds interconnectés. Dans ce cas de figure, imposer une unique vision globale est déraisonnable à la fois pour des raisons de performance due à la taille du modèle mais également pour des raisons d'isolation. Car un DDAS est le plus souvent composé de différentes partitions qui ont chacune une vision partielle, par exemple pour des problèmes de confidentialité. Dans l'exemple de la figure 4.4, la partie gauche peut être un ensemble d'environnement mobile tandis que la partie droite représente un *cluster*. Dans ce cas les nœuds de la partie gauche ne doivent pas voir ni adapter les nœuds de la partie droite et inversement. Pour cela il faut couper la vision du modèle et dans cette contribution cela fait partie la responsabilité de l'entité groupe. Un groupe peut donc *slicer* un modèle, c'est-à-dire le réduire à un sous-ensemble pour n'offrir qu'une sous-partie cohérente pour ses nœuds reliés. Ces groupes pivots interconnectent donc des DDAS séparés. Cette notion de *slicing* définie pour l'IDM [BCBB11] a été également exploitée de façon diverse pour les protocoles *Gossip* pour également obtenir un meilleur passage à l'échelle[JK06],[MZ08].

4.1.5 L'hétérogénéité des types d'adaptation

En plus de la divergence et des algorithmes de synchronisation, l'hétérogénéité des nœuds de calcul est un problème à prendre en compte pour la construction des DDAS. Comme annoncé dans les motivations, les nœuds de calcul peuvent avoir plusieurs types

donc chacun possède des spécificités concernant la puissance de calcul, le type de binaire supporté, etc. Chaque plate-forme exploite des primitives différentes afin de réaliser les adaptations nécessaires en fonction de ses spécificités. Par exemple un nœud de type Java va pouvoir exploiter un *class loader* pour le chargement à chaud de composants, tandis qu'un nœud embarqué va charger une librairie dynamique C. Pour aborder ce problème d'hétérogénéité, les solutions proposées dans l'état de l'art sont le plus souvent fondées soit sur un langage abstrait interprété de différentes manières suivant les plates-formes, soit sur une approche faite d'un *framework* complété par la plate-forme d'exécution. Dans les deux cas, ce gain d'homogénéisation se fait au prix de la perte des spécificités disponibles sur chaque plate-forme. Or dans le cas d'un système adaptatif les spécificités des plates-formes comportent entre autres leurs capacités d'adaptation, par exemple un nœud de type embarqué peut ne pas avoir les mêmes capacités de chargement de code à chaud qu'un *cluster* de serveur. Ces spécificités sont d'autant plus nécessaires dans l'approche Model@Runtime qui vise à valider *à priori* les modèles, ce qui nécessite de prendre en compte les capacités de chacun des nœuds afin de voir si le potentiel de mise à jour est réellement applicable.

Le modèle Kevoree propose à l'inverse de modéliser les nœuds mais surtout leurs différents types spécifiques qui sont composés de leurs différentes capacités abstraites d'adaptation. Dans ce modèle un nœud type contient donc un ensemble de primitives d'adaptation, le plus souvent représentant les opérations CRUD [BJ02] de ses entités (par exemple AjouterComposant, SupprimerConnecteur, etc). Un nœud type définit donc dans le modèle l'ensemble de ces primitives qu'il peut prendre en compte de manière abstraite tandis que l'implantation de ce nœud définit la sémantique concrète d'adaptation qui dirige une plate-forme spécifique (par exemple OSGi en Java). Voici une représentation schématique de ce modèle illustré par la figure 4.5.

FIGURE 4.5 – Représentation shématique de l'entité *NodeType* dans le modèle Kevoree

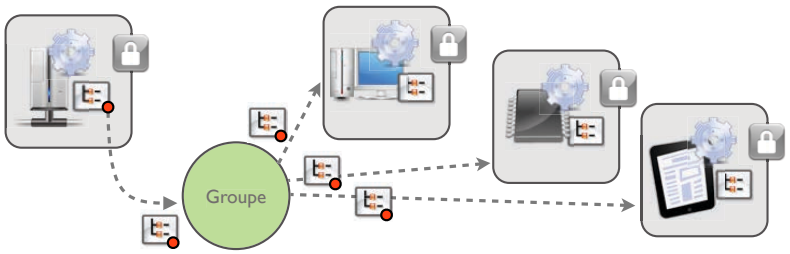
```
KevModel
|->Type id=NodeJava,version=1
    |-> Binaire URL=http://...
    |-> Primitives AjouterComposant,SupprimerComposant, DynLoadJAR
|->Type id=ComposantHello,version=1
    |-> Binaire nœudCible=NodeJava URL=http://...
|->Type id=NodeAndroid,version=1
    |-> Binaire URL=http://...
|->Nœud id=A type=NodeJava
    |-> Component id=CH1 type=ComposantHello.v1
|->Nœud id=B type=NodeAndroid
```

Un nœud possède une instance de Model@Runtime qui pilote alors les primitives définies dans le nœud type. A mi-chemin entre l'approche par DSL et par *framework*, Kevoree garde la capacité d'expression des spécificités de chaque plate-forme, et à la manière d'une API garde une séparation entre langage abstrait et interprétation, tout en gardant également l'API ouverte à de nouvelles primitives.

Chaque nœud est responsable de l'application de ces primitives concrètes et assure une encapsulation des ses composants et de sa plate-forme. Cependant les raisonneurs peuvent se servir des spécificités des plates-formes pour valider *a priori* les modèles mais

surtout en composant l'architecture en tenant compte des limitations de chacune. Par exemple il est possible de vérifier *à priori* que le composant CH1 possède une version binaire compatible avec le type de nœud hébergeant (*NodeJava*) et que la primitive AjouterComposant est présente. Une fois le modèle validé la propagation est homogène quel que soit le type grâce à la séparation des préoccupations préconisées. Sur la figure 4.6, le nœud le plus à gauche peut propager une mise à jour qui concerne un PC, un micro-contrôleur et une tablette sans pour autant connaître ni leurs primitives concrètes ni le protocole de communication qui est lui encapsulé dans le groupe.

FIGURE 4.6 – Mise à jour hétérogène



4.2 Propriétés attendues de la solution

Cette section synthétise les propriétés attendues d'une solution pour la construction des DDAS et notamment celui du cas sapeur-pompiers de motivation. Ces axes de conception donnent chacun lieu à une section de validation pour mettre en adéquation ces besoins vis à vis des résultats obtenus.

4.2.1 Séparation des préoccupations pour maximiser la réutilisation

Le modèle Kevoree poursuit donc le but d'être une abstraction qui explicite les éléments architecturaux d'un système distribué. Ce modèle s'attarde donc à modéliser de façon structurelle et non comportementale le système. Pour les mêmes raisons qui ont motivé les approches à composants, les éléments sont isolés en fonction de leurs préoccupations afin de maximiser leur réutilisation mais surtout pour faciliter le travail des outils d'assemblage et d'évolution d'architecture. En effet le modèle de délégation d'adaptation aux nœuds permet aux raisonneurs de s'affranchir de la connaissance des adaptations concrètes tandis que la délégation de la synchronisation des plates-formes aux groupes permet d'expliciter directement dans l'architecture les points de synchronisation. Ainsi le modèle Kevoree vise à rendre à la fois plus réutilisable et plus simple les différentes groupes et implantations de nœuds mais surtout les algorithmes de raisonnement qui ne sont plus « pollués » par ces préoccupations transverses. En tant qu'abstraction ce modèle permet donc à l'architecte de composer les DDAS en exploitant uniquement des concepts d'architecture.

4.2.2 Modèle extensible et versatile pour l'hétérogénéité des capacités de synchronisation et d'adaptation

La versatilité du modèle d'architecture doit permettre de représenter et d'encapsuler les différents algorithmes dans l'état de l'art pour la synchronisation des plates-formes *via* les groupes mais également la variabilité des capacités d'adaptation des différentes plates-formes. Afin de pouvoir s'exploiter dans un cycle de développement continu ce modèle doit s'accompagner d'opérateurs de mutation qui permettent d'exprimer les extensions des types du modèle mais surtout l'évolution des usages de ces types dans les plates-formes réelles.

4.2.3 Divergence de modèle pour la résilience aux fautes des nœuds

Le modèle Kevoree hérite des propriétés du Model@Runtime, à savoir une capacité à désynchroniser et resynchroniser sa représentation d'architecture afin de faire des processus de pré-déploiement. Cette divergence désormais possible entre le modèle d'architecture et les noeuds du système réel permet également d'effectuer les communications de façon plus opportuniste et moins synchrone. Tout en maîtrisant la divergence du système à l'aide d'un groupe, ce processus permet la distribution du modèle d'architecture et donc de la maîtrise du DDAS et assure ainsi sa résistance à la perte partielle des nœuds. Cette propriété sera donc évaluée sur un cas d'usage de nœuds mobiles qui se regroupent physiquement lorsque les liaisons radios de communication sont à portée et donc de façon opportuniste.

Chapitre 5

Modèle de composant étendu

5.1 Ensemble minimaliste des modèles CBSE

L'approche Model@Runtime proposée pour les systèmes distribués a besoin d'une entité manipulable à la fois pour diriger non seulement le développement mais également le déploiement. Les modèles de composants ont déjà prouvé leur efficacité pour la construction de systèmes adaptatifs large échelle sûrs grâce à de nombreux opérateurs de composition et moteurs de vérification (*model checking*). On trouve dans ce domaine beaucoup de travaux liés : Fractal, Frascati, OSGi, iPOJO pour ne citer qu'eux, la quasi totalité opèrent un rapprochement avec le paradigme de service pour forcer le découplage des définitions de ports de leur usage. D'un autre coté les *middlewares* à message tels que les implantations bus de la norme JMS ou AMQP ainsi que les *frameworks* à agent/acteur tel que Jade ou Akka ont montré que l'introduction de l'asynchrone directement dans le modèle de développement accompagné d'un modèle de concurrence permet des constructions qui en plus de pouvoir répondre à de fortes charges s'avèrent également efficaces dans le cas de réseaux maillés complexes. Le modèle de composants Kevoree se veut donc à la fois minimaliste afin de définir un sous-ensemble réutilisable des modèles existants et également étendu pour mixer les notions de composants et agent/acteur afin de préparer l'usage des composants en réparti. En d'autres termes, le modèle de composant Kevoree reprend les désormais classiques notions des composants, des ports et du cycle de vie, et y ajoute la notion de port de message asynchrone, un modèle de concurrence ainsi que la notion de canaux de communication (détaillée à la section 5.5). Ce modèle doit être suffisamment minimaliste pour pouvoir à la fois être appliqué aux travaux existants en ingénierie par composants et aux travaux sur les connecteurs réseaux et bus de message.

5.2 Type définition, type, Instance

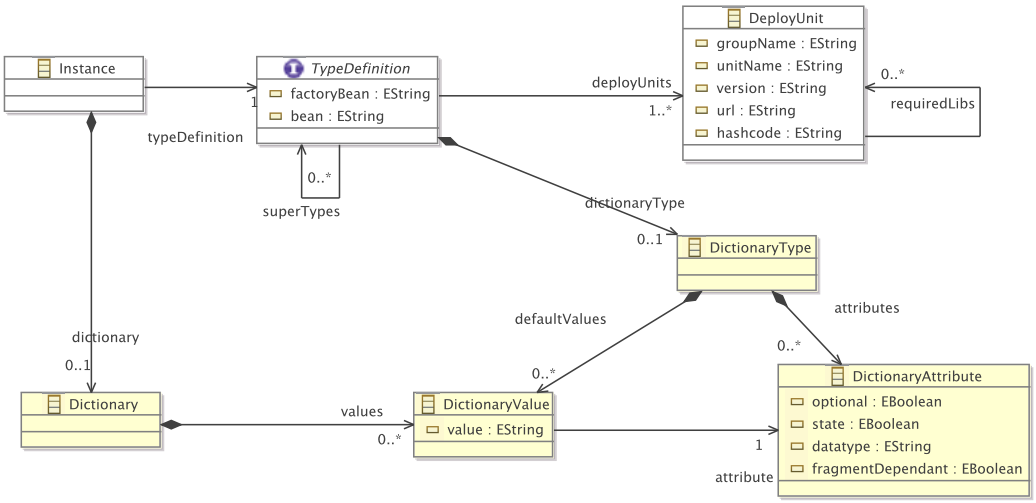
La séparation entre types et instances est essentielle pour la mise à jour d'un système de façon continue. En effet, d'une part, les définitions de type sont des artefacts réutilisables énonçant les fonctionnalités implantées et disponibles du système, et donc ce

que ce système est capable de réaliser. D'autre part, les instances représentent l'usage localisé de ces fonctionnalités et donc où ce système réalise cette fonctionnalité. Les définitions de type (*type definition*) doivent être distinguées des types généralement employés dans les langages à objets. Dans une approche de modélisation structurelle à composants comme celle de Kevoree les types correspondent au contrat extérieur du composant (ses ports, ses paramètres), contrat qui lui peut être inféré depuis le *type définition*. Dans le modèle Kevoree les types ne sont donc pas modélisés mais inférés, laissant libres leur interprétation et algorithme de vérification .

Les *types definitions* peuvent évoluer dans le temps : ils peuvent être mis à jour pour améliorer une fonctionnalité, simplement supprimés, ou de nouveaux *type definition* peuvent être introduits. Les activités de modélisation relatives au *type definition* sont généralement exploitées « offline » au moment de la conception ou de l'implantation, et synchronisées par la suite tandis que les instances sont eux des artefacts liés à la plate-forme et dont les mises à jour (paramètres, localisation) sont modélisées « online ».

Cette séparation type/instance qui suit un patron de conception bien connu [JW97] est donc une nécessité pour raisonner sur la substituabilité des fonctionnalités mais également pour connaître les instances concernées par une mise à jour de type. On peut faire l'analogie avec l'usage de ce patron dans les langages à objets qui séparent les définitions (classes) et les instances (objets). Cette analogie s'applique également aux paramètres des instances qui sont définis sous la forme d'un dictionnaire dont les entrées doivent respecter celles définies dans le dictionnaire type du type définition associé. Le dictionnaire d'une instance peut être à jour dynamiquement : on parle alors d'une mise à jour paramétrique.

FIGURE 5.1 – Type / Instance Kevoree Metamodel



Le patron type/instance modélise les deux entités dans le même niveau de modélisation MOF. Il est légitime de se poser la question d'exploiter ou non deux niveaux de modélisation pour cela ; la réponse vient surtout du fait que l'on a besoin des deux

informations dans le même « temps » de modélisation, il est nécessaire de connaître les instances pour savoir où déployer et dans le même temps il est nécessaire de connaître les types pour savoir quoi déployer. Cet usage n'est pas spécifique à Kevoree ; on le retrouve notamment dans les approches à composants dynamiques telles que SCA, qui définit une séparation entre *ComposantType* et *Composant*.

Ainsi le modèle Kevoree étend ce patron à l'ensemble de ses entités, canaux de communication, composants, nœuds, groupes de synchronisation. La figure 5.1 présente l'extrait du méta-modèle Kevoree qui illustre ce patron.

Héritage et *Type Definition*

A l'instar de l'héritage défini au niveau des classes dans les modèles à objets, le modèle Kevoree définit une notion d'héritage pour les *type definitions*. Cet héritage peut être multiple. Il s'exprime par la relation *superTypes* d'un *type définition* vers ses parents. La sémantique de cette relation est spécialisée par les différents éléments concrets qui en héritent (*component*, *channel*, *node*). Cependant elle assure de façon générique une garantie de conformance descendante des fonctionnalités des sous-types. Par exemple, le contrat d'un composant type héritant d'un autre doit au moins contenir les propriétés et les ports du parent.

Cette relation assure un premier niveau de réutilisation et composition dans le modèle qui peut être étendu dans les sous-éléments, par exemple avec des composites de composants types. Tout comme les classes objets, l'intérêt de faire une relation au niveau type et non instance permet d'assurer un fonctionnement cohérent de toutes ses instances, cette propriété est notamment nécessaire pour assurer le fonctionnement cohérent sur différents noeuds d'instance du même type.

5.3 Cycle de vie

Les instances déployées dans les plates-formes Kevoree suivent un cycle de vie dont le comportement est défini dans le *type definition*. Ainsi comme l'illustre la figure 5.3 les *types definitions* spécifient un comportement pour les étapes *Start*, *Stop* et *Update*. Ces méthodes sont appelées suivant l'automate à état illustré en figure 5.2, qui définit qu'une instance peut être démarrée (méthode *Start*), arrêtée (méthode *Stop*), et que ses paramètres peuvent être mis à jour après le démarrage (méthode *Update*). Dans les trois cas l'instance est mise en pause (par exemple les ports d'entrées ne traitent plus les messages pour un composant) pour garantir la non concurrence ; ceci sera détaillé par la suite. Ce cycle de vie correspond à celui communément admis dans la communauté CBSE, il est notamment similaire à celui de Fractal/Frascati ainsi qu'à OSGi. L'étape *Update* est quant à elle plus spécifique, elle trouve son équivalent dans le modèle Fractal par les évènements émis par les membranes des composants, dans le modèle iPOJO également Cette étape n'a pas d'équivalent en OSGi, qui n'a pas de mise à jour paramétrique.

FIGURE 5.2 – Cycle de vie des instances Kevoree, machine à état

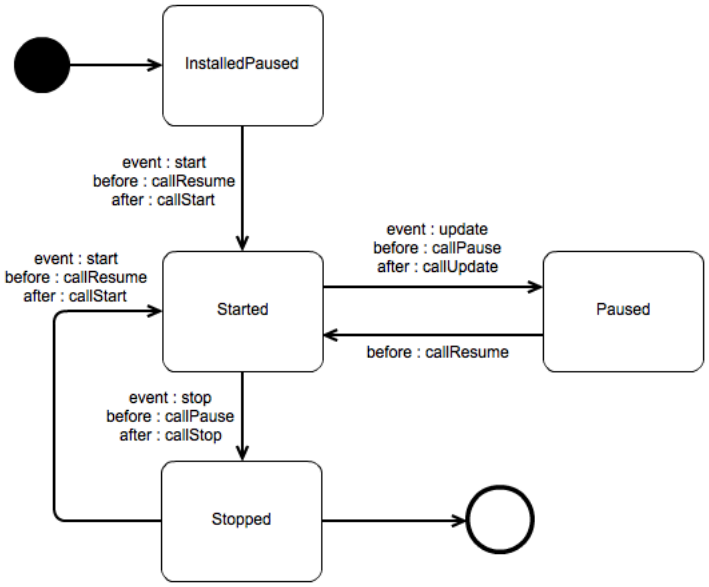
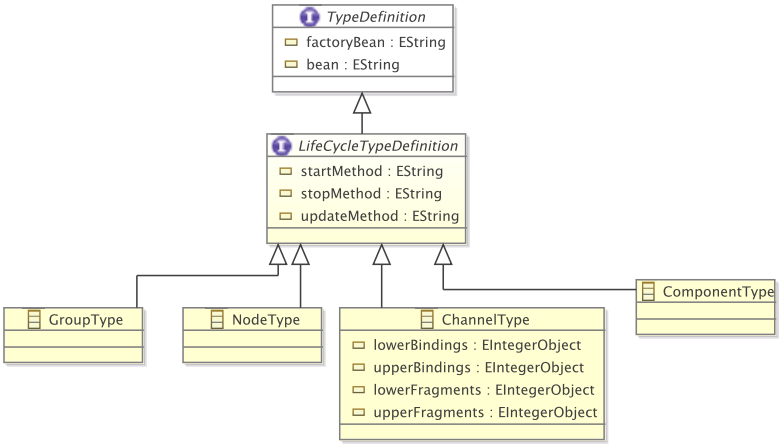


FIGURE 5.3 – Cycle de vie des instances Kevoree, extrait du méta-modèle



5.4 Composants

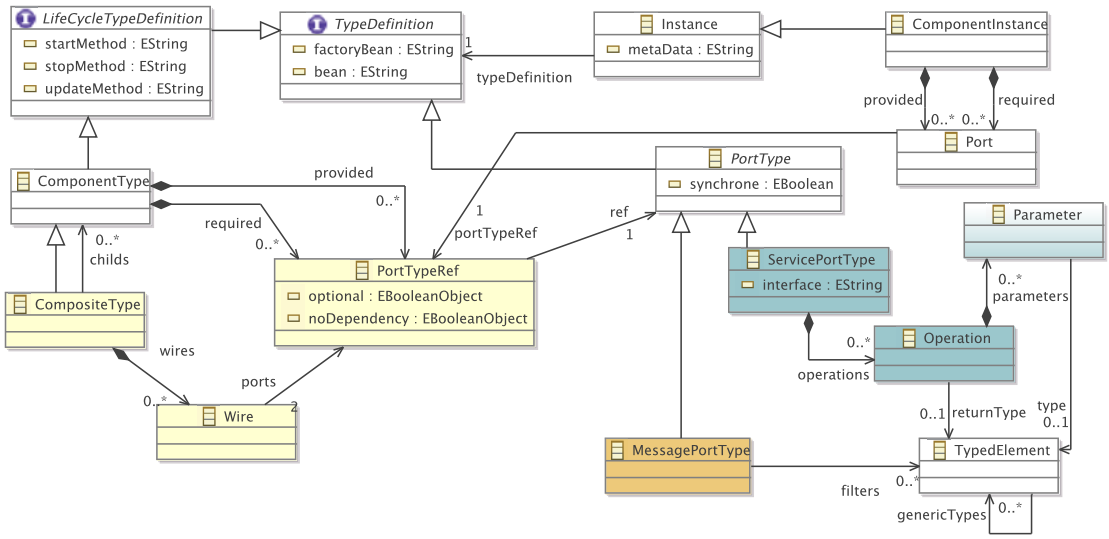
Comme introduit dans la section 5.1, le modèle de composant en lui-même reste aligné sur les travaux précédents. Un composant est défini comme une entité close garantissant une fonctionnalité si son contrat est respecté. Le terme « clos » signifie ici que ni le cycle de vie d'un composant ni ses processus internes ne doivent être dépendants d'autres ressources ni d'autres processus. Ce contrat structurel est défini par

des ports, seules interfaces de communication permettant les échanges avec l'extérieur au moyen d'interconnexions. Le contrat du composant définit donc un ensemble de ports requis nécessaires à son fonctionnement interne ou optionnels, et un ensemble de ports offerts aux autres composants. Comme le reste des éléments du modèle les composants suivent le patron type/instance, les composants instances sont donc composés de ports connectables tandis que les composants types sont quant à eux composés de *PortType*. L'extrait du méta-modèle qui représente les composants types est illustré en figure 5.4.

Les ports types ont été étendus par rapport aux travaux précédents pour définir non seulement les types de données échangés à la manière des services mais aussi pour y inclure les types de patron d'échanges (synchrone/asynchrone). Le méta-modèle Kevoree définit deux familles de types de ports : *ServicePortType* et *MessagePortType* détaillés respectivement en section 5.4.1 et 5.4.2.

En terme d'opérateur de composition, les composants types peuvent exploiter l'héritage de *type definition* de Kevoree afin de réutiliser les définitions de ports. Au niveau instance il est également possible d'exploiter un modèle de composite aligné avec le méta-modèle SCA.

FIGURE 5.4 – Extrait du méta-modèle de composant



5.4.1 ServicePortType

Le port type *Service* définit un port capable d'effectuer des appels classiques de service qui respectent un principe de RPC. Ce port est typé et fournit son contrat d'interface sous forme d'un ensemble de méthodes avec paramètres d'entrée et de retour. Le patron d'échange de message (Message Exchange Pattern (MEP)) est ici un *synchronous InOut*. En d'autres termes les appels sur les ports service sont synchrones, s'apparentent

à un appel de méthode dans les langages à objets et fournissent systématiquement un résultat.

5.4.2 MessagePortType

Les ports de type *MessagePortType* sont dédiés à l'envoi et à la réception de structure de message de façon asynchrone. A l'aide de ses ports le modèle Kevoree peut proposer un modèle évènementiel en addition du classique modèle RPC. Le MEP est donc ici un *asynchronous InOnly*, c'est-à-dire que les ports ne fournissent pas de résultat en retour. Les ports de type *MessagePortType* sont typés ou non. Le type d'un port message est caractérisé par l'ensemble des messages évènement qu'il va accepter de traiter.

5.4.3 Modèle de concurrence des ports

Le contrôle des accès concurrents à un port de communication et donc à une fonctionnalité métier peut être à la fois considéré comme une problématique transverse mais également lié si le composant doit assurer les propriétés ACID. Dans le cas présent, l'ensemble des ports de communication est fondé sur des échanges de messages (bidirectionnel pour le RPC), le contrôle des accès concurrents vérifie qu'aucun port ne traitera deux messages du même port « dans le même temps' ». Plusieurs approches sont disponibles dans la littérature pour vérifier structurellement cette non concurrence : les approches défensives à base de verrou et zone exclusives, les approches à acteurs ou les approches à zone de mémoire transactionnelle, qui résolvent ce problème au niveau « état ». Dans tous les cas, la gestion de cet accès concurrent est un problème difficile pour le concepteur d'application et plus encore lorsque cette problématique est disséminée de façon différente dans toute l'application. Comment dès lors assurer que l'arrêt d'un processus ne va pas mettre à mal l'ensemble de la synchronisation ou conduire à un inter-blocage ?

Dans une vision IDM et afin de garantir une séparation des préoccupations le modèle Kevoree déporte la gestion de ce contrôle d'accès à l'extérieur de la définition des fonctionnalités. Ainsi le développeur écrit son code métier en décorant les ports avec les propriétés de concurrence attendues et pour lesquelles le code glue sera généré.

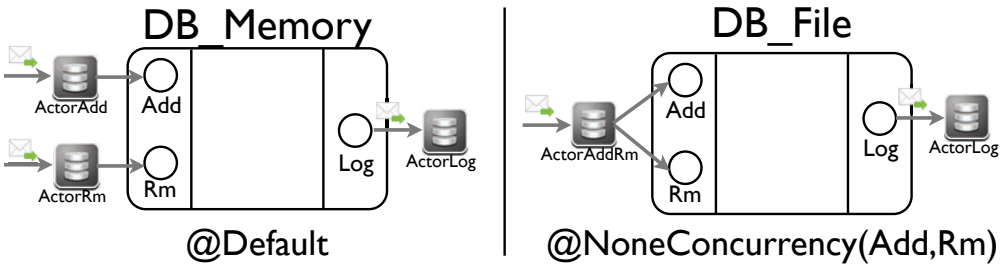
Le modèle des acteurs définit une notion de processus léger échangeant des messages non mutables. Le modèle de traitement définit chaque processus comme ayant une file d'attente (boîte aux lettres) dans laquelle s'empilent les messages, qui sont dépilés séquentiellement pour traitement. Ce modèle a déjà prouvé ses performances dans des projets comme Akka ou les frameworks à agent. L'idée générale du modèle de concurrence de Kevoree est donc de *mapper* les ports sur des Acteurs pour lesquels la sémantique est très proche.

L'intérêt est triple. Premièrement, le cycle de vie des acteurs est clairement défini et est proche de celui des instances Kevoree. Sa généralisation à tous les ports donne une homogénéité à l'application qui permet de prévoir l'état de chaque port afin de choisir le meilleur moment pour réaliser les adaptations sans perturber le fonctionnement. Deuxièmement, cette approche *lock-free* permet de réduire la synchronisation si

coûteuse sur des machines multi-cœurs et surtout si délicate dans le cas de systèmes adaptatifs. En effet, il est très difficile d'adapter un programme qui pose des verrous car il est alors impossible de savoir si on ne risque pas d'entraîner un inter-blocage, par exemple en déplaçant un composant et en arrêtant un *thread* qui devait en réveiller un autre. Troisièmement, le modèle des acteurs est fondé sur le modèle des files d'attente de type *FIFO*. Ce modèle est notamment facile à composer afin de calculer les modèles comportementaux. Ainsi l'adoption de ces files comme points de synchronisation permet de calculer les modèles globaux d'échange et permet ainsi l'usage des Model@Runtime comportementaux définis par Cheng *et al* [ZGC09].

Par défaut le *mapping* entre les processus légers des acteurs est donc du type *un pour un*, c'est-à-dire un acteur par port requis ou fourni. Ceci est illustré par la figure 5.5, chaque cadre représente un composant, les ports requis sont dessinés à droite , les ports fournis à gauche. Ainsi dans l'exemple DB_Memory les ports nommés *Add* et *Rm* peuvent être appelés de manière indépendante et concurrente, par contre les appels à chacun des ports suit un ordre séquentiel. L'acteur empile donc les messages et appelle les méthodes (par exemple dans le cas de Java) des ports suivant l'ordre du *Scheduler*.

FIGURE 5.5 – Injection d'acteur devant les ports de composant Kevoree



Le modèle Kevoree permet également d'affiner le *mapping*, ainsi dans le second exemple il est possible de déclarer une propriété de non concurrence entre les ports *Add* et *Rm*. Ainsi un seul acteur est généré, faisant le *dispatch* entre ces ports. Dans le deuxième exemple de la figure 5.5 le composant DB_File se protège des accès concurrents sur ces deux ports car il manipule une ressource qui n'accepte pas les écritures concurrentes. Un acteur est également injecté sur les ports de sortie de sorte que l'ensemble des entrées et sorties se fassent dans des processus légers (en opposition aux *threads*) du composant, respectant ainsi la définition première des composants d'entité close (isolé d'un point de vue processus). Pour que cette isolation soit complète et pour respecter le paradigme d'acteur il faudrait assurer structurellement l'immutabilité des messages ou définir une stratégie défensive de copie ou au contraire exploiter une stratégie *zero-copy*. Ce choix est laissé à la charge du concepteur de composant dans son contrat, afin de générer le code le plus adapté.

Le modèle théorique d'échange de message des acteurs peut être implanté de plusieurs manières. Ainsi par exemple pour la plate-forme Java l'implantation de Kevoree possède un mapping de cette sémantique vers le framework d'acteur de Scala, ou encore en exploite les files d'attente partagées de Java et enfin le modèle de buffer circulaire

du framework Disruptor [1] [2].

5.5 Canaux de communication : channels

5.5.1 Des connecteurs aux *middlewares* : motivation pour une modélisation d'architecture de la communication

Les modèles de composant s'accompagnent le plus souvent dans la littérature d'un modèle d'architecture qui les interconnecte afin de les faire collaborer et fournir aux ports requis les accès aux ports fournis. Mais au-delà d'un simple *binding* entre deux ports cette interconnexion peut être complexe et diverse (plusieurs possibilités) notamment dans le cas où les composants sont sur deux nœuds de calcul différents. De même lorsqu'il s'agit de faire collaborer plus de deux ports une sémantique de distribution est implicitement nécessaire (*broadcast*, *unicast*, etc). De manière consensuelle par la communauté scientifique, ce problème est identifié depuis les débuts des modèles à composants sous la notion de connecteurs. La classification en 2000 par Medvidovic *et al* [MT00] définit d'ailleurs les caractéristiques de ces connecteurs et identifie déjà le manque de leur modélisation comme une entité de première classe dans les modèles de composants. En effet soit inexistantes dans beaucoup de modèles, ou implantées sous forme de couche de transport implicite (EJB2, DOSGi, iPOJO), ou encore cantonnées à des *bindings* 1-1 (SCA) de type RPC, les capacités de déploiement dynamique et la réutilisation de ces entités sont souvent faibles. Plus récemment, ce constat est encore partagé dans la thèse d'Olivier Barais [Bar05], la synchronisation et communication apparait également dans la taxonomie de Beugnard *et al*[BJPW99] en 1999 comme une nécessité pour les contrats de composants.

Paradoxalement les *middleware* capables de définir des couches transport ont quant à eux défini des sémantiques riches ainsi que des langages de descriptions (IDL) dynamiques. Des DSL comme Apache Camel définissent alors des notions de route complexe exprimant des échanges entre des *Endpoint*. De la même façon les bus de message définissent des sémantiques de *Publish and Subscribe* ou *distribution queue*. Classiquement ces *middleware* accompagnent les plates-formes à composants pour effectuer la tâche de transport, on retrouve même un standard de ce couplage avec JBI qui définit un bus de message NMR couplé à chaque nœud de calcul et distribuable. Mais il y a alors un fort manque d'explicitation de ces interactions dans le modèle d'architecture, ce qui complexifie toutes les approches réflexives en général, et surtout les mises à jour et déploiements sur systèmes distribués.

En 2000 encore, Beugnard *et al* [Beu00] imaginent une notion de médium capable d'expliciter plusieurs sémantiques de communication en définissant des *multi-middleware* entre des ports de composants distribués, capables alors au déploiement de se séparer en bouchon de communication sur chaque nœud les utilisant.

En 99, Guerraoui [Gue99] insiste également sur ce besoin de séparation entre la communication et les éléments de l'application métier distribué. Dans ce papier il est

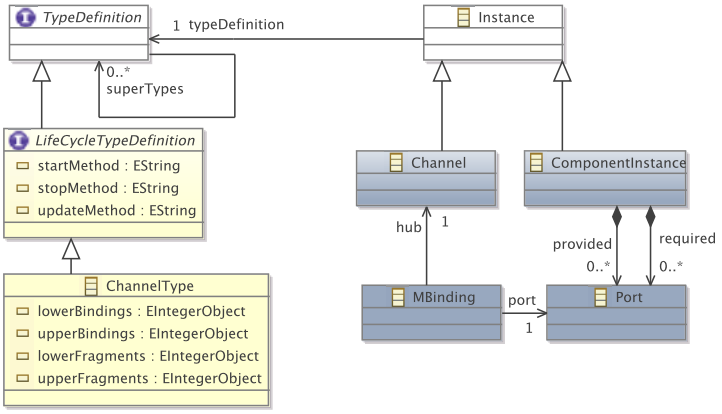
1. https ://github.com/LMAX-Exchange/disruptor
2. http ://disruptor.googlecode.com/files/Disruptor-1.0.pdf

question de placer la communication comme une entité de première classe et surtout flexible (non assemblée au moment de la compilation) vis à vis des composants, car je cite : “il est difficile de prévoir quel mécanisme de distribution sera le plus adapté pour une application”.

5.5.2 Définition des channels

Le modèle Kevoree définit donc une entité nommée *Channel* responsable de la modélisation de la sémantique d'échange entre plusieurs ports de composant potentiellement distribués. Inspirée à la fois des connecteurs, médiums[MB05],[PKBGS08] et *micro-middleware*, cette entité est de première classe dans le modèle, au même niveau que les composants. Les *channel* ont pour vocation de pouvoir expliciter dans le modèle d'architecture, à la fois des échanges de type service RPC mais également des sémantiques d'échange de messages complexes inspirées des algorithmes distribués, tels que : les échanges après élection, les files d'attentes distribués, etc. La figure 5.6 illustre cette définition par un extrait du méta-modèle.

FIGURE 5.6 – Extrait méta-modèle Kevoree : ChannelType

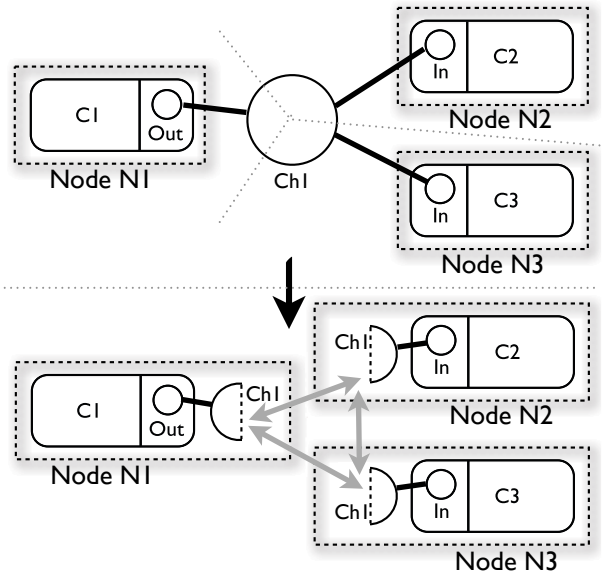


Ainsi les *ChannelType* définissent la sémantique d'échange tandis que les *Channel* définissent une relation entre un ensemble de ports par le biais de *bindings*. Ces *bindings* représentent l'abonnement' d'un port à un canal de communication en entrée ou sortie et sont nécessairement simples (relation 1-1) car les sémantiques de distributions complexes sont contenues dans le *Channel* lui-même. Comme les autres instances de Kevoree les *Channels* ont un *type définition* et un cycle de vie, ceci répond donc au besoin de réutilisation et d'extensibilité identifié par Medvidovic, ainsi qu'à la capacité de déploiement dynamique identifiée par Barais et Beugnard.

Un *channel* n'est pas localisé, c'est-à-dire qu'il n'est pas contenu de manière analytique dans un nœud de communication. A l'inverse, ses localités sont définies par ses relations avec les ports des composants modélisés par les *MBindings* , qui eux sont hébergés sur des nœuds de calcul. Ainsi si un *channel* est une seule entité pour l'analyse et la conception, il se sépare en fragments pour chacune de ses localités, c'est-à-dire

des nœuds ayant au moins une relation avec lui. Cette séparation au déploiement est illustrée par la figure 5.7. Dans cette figure le *channel* *Ch1* lié aux composants *C1* , *C2* et *C3* , respectivement hébergés sur les nœuds *N1* , *N2* et *N3* , est fragmenté en autant de fragments que de nœuds et ses fragments sont liés par des liaisons internes.

FIGURE 5.7 – ChannelType fragmentation



Dans ce modèle, les *Channel* ont la responsabilité des *bindings* locaux et transmettent *via* les communications inter-fragments si une communication vers un port distant est nécessaire. De cette façon le modèle Kevoree assure structurellement qu'un *binding* n'héberge pas de sémantique complexe.

Un *ChannelType* est donc caractérisé par ses capacités de gestion de *binding* locaux et distants. Cette propriété se retrouve sur la méta-classe *ChannelType* pour laquelle une borne inférieure et supérieure est modélisable. Les algorithmes contenus dans les *Channel* sont bien des algorithmes distribués comme l'avait préssenti Beugnard. En effet ils doivent composer avec les échanges entre les fragments pour mettre en œuvre leur sémantique (par exemple propagation avec ordre total ou partiel, rejeu ou non etc ...).

Avec ces outils de fragments, il est possible de construire les standards de communications tels que les clients serveurs, les files d'attentes distribuées, les topics d'abonnements distribués, etc ...

5.5.3 Pourquoi une nouvelle entité et non des composants dédiés ?

Ce choix se justifie surtout pour maximiser la réutilisation, en effet par définition les composants et les *Channel* ont une portée de visibilité très différente. Afin d'assurer la propriété de processus clos, les composants ne doivent pas voir leurs *bindings*. À l'inverse, afin de pouvoir écrire les algorithmes distribués les *Channel* voient les *bindings* locaux

et distants (par exemple pour connaitre le domaine de nœud dans le cas d'un algorithme de type *gossip*). Mixer les préoccupations de code métier et de communication dans la même abstraction obligerait donc à altérer la sémantique des composants et les rendrait moins réutilisables.

5.5.4 Modèle de concurrence

A l'instar des composants, les *Channel* nécessitent un modèle de concurrence explicite afin de pouvoir écrire les échanges entre les différents fragments. Le modèle d'acteur y est également adapté, car les algorithmes doivent ordonnancer des appels concurrents venant à la fois des *bindings* locaux mais également des autres fragments. Le *mapping* est cependant encore plus direct, et chaque fragment suit une sémantique d'acteur. Un fragment possède donc une seule file d'attente pour les appels locaux et distants. Le *Channel* protège ainsi son cycle de vie et peut être également mis en pause lors des modifications de ses *bindings*.

Chapitre 6

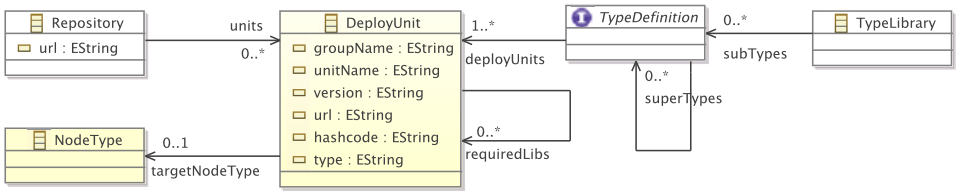
Model@Runtime distribué

Ce chapitre détaille la partie principale de la contribution qui s'articule autour d'une extension du Model@Runtime pour les systèmes distribués adaptatifs continus. Cette extension comprend notamment l'ajout d'un modèle de déploiement, la définition d'un cœur de gestion de plate-forme dirigé par les modèles ainsi que la modélisation des nœuds et de leur capacité de synchronisation selon le concept de *groupes*.

6.1 Modèle de déploiement

Comme annoncé précédemment, la conception continue justifie la nécessité d'une relation entre le modèle de déploiement et celui de design afin de pouvoir passer du modèle d'assemblage aux plates-formes d'exécution. Ce modèle d'assemblage extrait les informations des artefacts de développement pour assurer la traçabilité des binaires avec les fonctionnalités du système. Dans l'approche proposée les fonctionnalités sont modélisées par les définitions de type tandis que la notion de binaire est exprimée à l'aide des *DeployUnits* . Le modèle Kevoree intègre ces deux notions dans un méta-modèle commun afin de pouvoir exploiter la même source de modèle pour le déploiement et la conception du système, mais aussi et surtout pour exprimer les liens de dépendance entre elles. En effet, d'un côté les définitions de types qui représentent soit des composants soit des canaux de communication ou encore des nœuds types dépendent d'un ensemble de *DeployUnits* nécessaires pour leurs l'implantation. De plus les *DeployUnits* possèdent quant à elles une notion de plate-forme cible et donc une relation avec les nœuds types afin de modéliser la notion de comptabilité des binaires. Les binaires sont des entités hébergées sur un serveur dédié à leur stockage suivant le processus défini à la figure 4.2, cette relation s'exprime *via* la notion de *Repository* dans le modèle. La figure 6.1 représente un extrait du méta-modèle illustrant ces relations, qui sont détaillées dans des sous-sections dédiées ci-dessous.

FIGURE 6.1 – Modèle de déploiement



6.1.1 Modèle de DeployUnit

La notion de *DeployUnit* est inspirée des modèles de déploiement actuels que l'on peut trouver soit dans les outils de développement tels que Apache Maven [1] ou encore Apache Ivy [2] pour ne citer qu'eux, mais également dans les systèmes de paquets qui servent au déploiement dans les systèmes linux tel que RPM [3] ou encore les fichiers DEB [4]. Cette notion désigne donc de la même manière à ce qui existe dans l'état de l'art, une entité déployable et identifiable de manière unique sur un serveur. Celle-ci doit être auto-contenue hormis un ensemble de dépendances explicitement représentées. Les *DeployUnit* et leurs dépendances sont téléchargeables depuis un *Repository* .

6.1.1.1 Relation de similarité des unités de déploiement

Les unités de déploiement sont identifiables par deux relations distinctes. D'un coté l'identification de l'unité et donc de la fonctionnalité et des définitions de types contenus se fait par agrégation des noms de *Groupe* , *Artefact* et *Version* . L'ensemble DU des unités de déploiement se définit de la façon suivante :

Déf 2. $DU \rightleftharpoons \forall x.(x.type = "DeployUnit")$

La relation suivante exprime alors la similarité des unités :

Déf 3. $(x \equiv y) \rightleftharpoons (\forall x.\forall y.(x \in DU \wedge y \in DU \wedge x.groupID = y.groupID \wedge x.artefactID = y.artefactID \wedge x.version = y.version))$

À cette notion d'identification s'ajoute une notion de code de hachage du binaire considéré, servant à valider l'absence de modification de ce dernier. Cette notion est particulièrement utile dans le cas de la conception continue, qui peut faire évoluer les implantations tout en gardant la même identification (par exemple dans le cas de corrections de bugs). Une unité est donc considérée comme étant mise à jour si elle est équivalente en nommage à la précédente utilisée et si son code de hachage diffère. Cette relation d'égalité stricte s'exprime alors de la façon suivante :

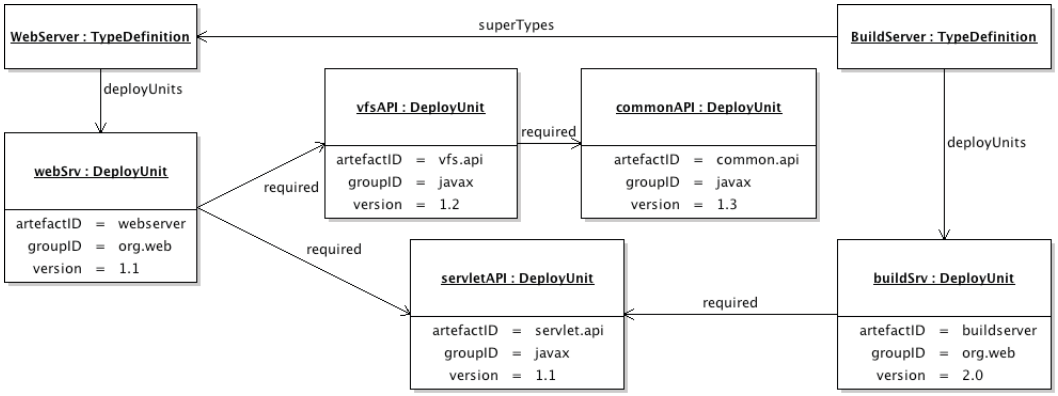
Déf 4. $(x = y) \rightleftharpoons (\forall x.\forall y.(x \in DU \wedge y \in DU \wedge x \equiv y \wedge x.hash = y.hash))$

1. http ://maven.apache.org/
2. http ://ant.apache.org/ivy/
3. http ://en.wikipedia.org/wiki/Rpm_(file_format)
4. http ://en.wikipedia.org/wiki/Deb_(file_format)

6.1.1.2 Graphe de dépendance

Le modèle présenté en figure 6.1 définit des relations de dépendance entre unités de déploiement. Chaque définition de type dépend donc d'un ensemble de *DeployUnit* qui elles-mêmes possèdent un ensemble de dépendances tierces qui pointent également vers des *DeployUnit* . Cette modélisation représente donc les binaires sous forme d'un graphe de dépendance dont les points d'entrées sont les *TypeDefinition* . Ce graphe est illustré par un exemple dans la figure 6.2, qui présente un *TypeDefinition* Web serveur et un serveur d'intégration avec leurs dépendances.

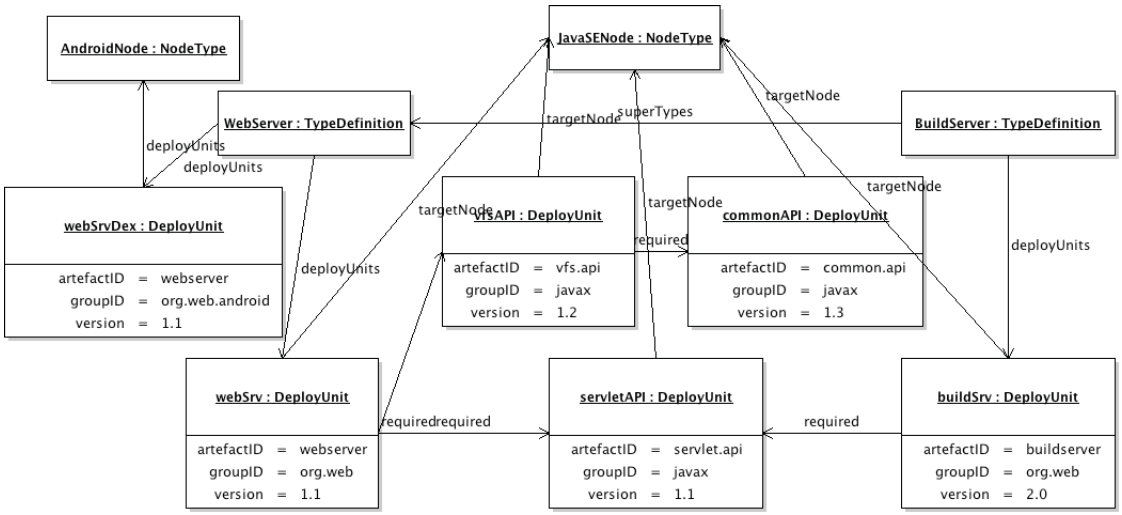
FIGURE 6.2 – Graphe de dépendance



6.1.1.3 Hétérogénéité des binaires

Un *TypeDefinition* est une fonctionnalité qui peut être implantée dans des environnements hétérogènes. Si l'implantation doit respecter le même comportement d'une plate-forme à l'autre, son graphe de dépendance peut lui être différent. Pour modéliser cette hétérogénéité inhérente aux compilations croisées, le modèle Kevoree permet d'associer plusieurs *DeployUnit* à un type et permet en sus d'ajouter une relation entre les unités de déploiement et les nœuds types qui représentent des plates-formes. Cette relation permet de construire la relation de conformité entre un binaire et une plate-forme. La figure 6.3 étend la figure 6.2 en ajoutant ces informations de conformité, ainsi le type *WebServeur* définit une implantation compatible Android à l'aide d'une unité ayant pour *targetNode* Android alors que le serveur de build n'est lui compatible qu'avec les plates-formes Java, car ne possède aucun *DeployUnit* dans son ensemble d'implantation compatible. Il est à noter que l'unité de déploiement *WebServer* n'a pas le même graphe de dépendance dans sa version pour Android et par rapport à celle de Java. Le modèle est volontairement ouvert à ces cas de figures pour prendre en compte la variabilité des implantations et des librairies tiers.

FIGURE 6.3 – Graphe de dépendance hétérogène



6.1.2 Résolution et sélection de binaire

Le déploiement d'un *TypeDefinition* sur un noeud de calcul nécessite de calculer l'ensemble minimum de binaires à déployer pour son fonctionnement. Cette sous-section définit les relations de conformité et les opérateurs de sélection pour cette opération.

6.1.2.1 Relation de conformité par plate-forme

Une définition de type est compatible et donc déployable sur une plate-forme x s'il existe dans le modèle au moins une implantation dans ses unités de déploiement directes compatible avec cette plate-forme d'hébergement x. L'ensemble NTY des types de nœud se définit de la façon suivante :

Déf 5. $NTY \rightleftharpoons \forall x.(x.type = "NodeType").$

Ainsi que l'ensemble TD des *types definitions* :

Déf 6. $TD \rightleftharpoons \forall x.(x.type = "TypeDefintion").$

Suivant la relation suivante, un *TypeDefinition* est conforme à un nœud type si une de ses unités de déploiement est prévue pour un nœud type ayant une relation d'héritage avec le nœud hébergeant :

Déf 7. $compat : (x, n) \rightarrow Bool \rightleftharpoons (\forall du.\forall x.\forall n.(x \in TD \wedge n \in NTY \wedge du \in DU \wedge \exists du.(du \in x.deployUnits \wedge n <: du.targetNode)))$

Cette relation comporte un opérateur $<:$ de sous-typage entre les nœuds types. En effet, tout comme les *TypeDefinition* du modèle Kevoree , les nœuds types peuvent

définir des relations d'héritage. Cette relation assure dans le cas des nœuds la compatibilité binaire descendante, ainsi un nœud héritant d'un autre hérite également de sa capacité d'interprétation des artefacts de déploiement. Un nœud type est un sous-type d'un autre s'il apparaît dans la hiérarchie définie comme suit :

Déf 8. , $(n1 <: n2) \rightleftharpoons (\forall n1.\forall n2.(n1 \in NTY \wedge n2 \in NTY \wedge (n2 \in n1.superTypes \vee \exists np.(np \in n1.superTypes \wedge np <: n2))))$

6.1.2.2 Sélection de dépendance nécessaire par plate-forme

L'héritage ne s'applique pas uniquement aux nœuds types mais à l'ensemble des *types definitions* qui sont eux-mêmes déployés sur les nœuds. Cette relation d'héritage définit implicitement une relation de dépendance. En effet afin de fournir des fonctionnalités au moins équivalentes les *types definitions* qui dépendent d'un autre héritent et agrègent également leurs dépendances définies suivant la relation suivante :

Déf 9. $UsedDU_{x,n} \rightleftharpoons (\forall x.\forall n.(x \in TD \wedge n \in NTY \wedge du \in DU \wedge (du \in x.deployUnits \wedge compat(x,n) \vee \exists superT.(superT \in x.superTypes \wedge du \in UsedDU(superT,n)))))$

Dans cette relation l'ensemble des dépendances nécessaires pour un *type definition* est donc l'agrégation des toutes les dépendances compatibles. Cependant un *type definition* pouvant exister dans de multiples versions il est nécessaire d'assurer un ordre de résolution pour éviter les collisions. Cet ordre se fonde sur la similarité la plus proche dans l'arbre d'héritage. En d'autres termes on choisit l'implantation la plus proche du type du nœud d'hébergement. Par exemple si un nœud type Java est hérité par un JavaEtendu et qu'un composant type est disponible pour les deux implantations, dans le cas d'un déploiement sur JavaEtendu on prendra la version JavaEtendu de manière systématique. Ceci donne la spécialisation de la relation de comptabilité suivante :

Déf 10. $compat : (x,n) \rightarrow Bool \rightleftharpoons (\forall du.\forall du2.\forall x.\forall n.(x \in TD \wedge n \in NTY \wedge du \in DU \wedge du2 \in DU \wedge \exists du.(du \in x.deployUnits \wedge n <: du.targetNode) \wedge \nexists du2.(du2 \in x.deployUnits \wedge du.targetNode <: du2.targetNode)))$

6.2 Modèle de Nœud : conteneur d'instances et sémantique d'adaptation

Les entités nœuds sont donc la pierre angulaire du modèle Kevoree . Cette notion représente les nœuds de calcul des systèmes distribués et plus précisément dans le cas des DDAS représente les conteneurs capables d'héberger dynamiquement des instances (par exemple des composants). L'approche proposée cherche à répondre aux besoins exprimés dans l'état de l'art dans le cas de DDAS hétérogène, à savoir l'explicitation des différentes plates-formes d'adaptation tout en gardant l'extensibilité nécessaire pour l'intégration de nouvelles sémantiques d'adaptation. La section suivante détaille la sous-partie du modèle Kevoree relative aux nœuds, ainsi que les mécanismes de composition proposés pour faire face à leur diversité. Est également abordée ici la définition des

services que le nœud doit offrir localement au *Model@Runtime* pour que ce dernier puisse piloter le DDAS de façon globale.

6.2.1 Nœud Kevoree : pilote d'adaptation de plate-forme pour un processus Model@Runtime

En d'autres termes, les nœuds désignent une entité capable de piloter une plate-forme concrète dont ils ont la responsabilité et sur laquelle ils ont un accès exclusif. Ce pilotage est dirigé par un processus M@R local qui ordonnance l'exécution de code pour synchroniser la plate-forme suivant les différents modèles qui lui sont proposés. Un nœud de calcul concret du système est donc l'association d'une instance de nœud Kevoree , d'une plate-forme d'exécution et d'adaptation et d'un moteur d'exécution *Model@Runtime*. Cet assemblage est illustré par la figure 6.4 qui représente par une double flèche (1) les données échangées entre le *Model@Runtime* et le nœud instance. Les données échangées en (1) sont relatives uniquement au modèle d'architecture Kevoree . Le lien entre le nœud instance et la plate-forme (2) représente lui des accès natifs d'adaptation de la plate-forme, par exemple pour créer concrètement des composants. De façon simplifiée, le nœud fait donc l'interface entre les abstractions du modèle et le code exécutable spécifique pour la plate-forme.

FIGURE 6.4 – nœud instance de plate-forme

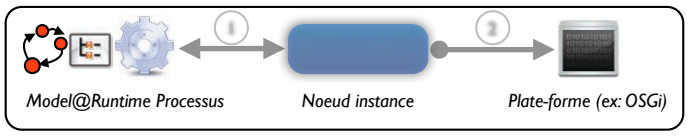


Plate-forme piloté par Model@Runtime

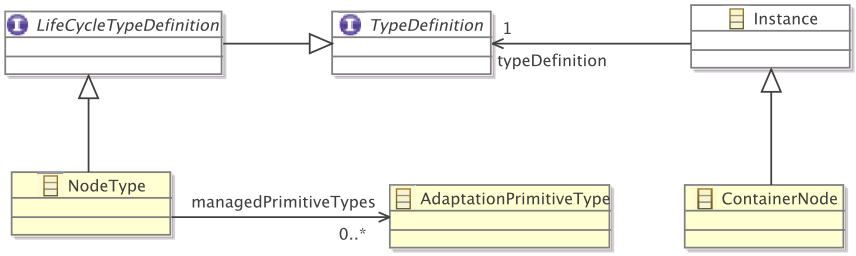
Les nœuds sont donc spécifiques pour le pilotage d'une plate-forme dédiée et surtout n'ont qu'une responsabilité locale, c'est-à-dire qu'ils n'appliquent que du code ayant un impact sur la plate-forme fille du nœud instance et non sur des nœuds distants. Un nœud est un conteneur qui assure donc l'isolation de son exécution et de ses accès. Pour les mêmes raisons d'extensibilité que les composants, la notion de nœud suit le même patron type/instance que les autres entités du modèle Kevoree . Ainsi le nœud désigne une plate-forme concrète mais sa sémantique d'adaptation est elle définie dans un nœud type associé. Un ensemble de *NodeType* permet alors de modéliser la sémantique d'adaptation des différentes plates-formes d'exécution d'un DDAS hétérogène. La figure 6.5 suivante illustre ces propos par l'extrait du méta-modèle Kevoree correspondant :

Les *NodeType* sont dédiés au service des demandes d'adaptation du M@R : ils doivent lui fournir les services pour faire évoluer le système à travers eux, d'un état vers un autre en exploitant uniquement des modèles. Pour cela, ils doivent fournir :

– l'opérateur qui extrait en terme d'actions la différence entre deux modèles ;
– l'opérateur qui permet d'appliquer les actions et donc les transitions.

Le nœud doit fournir un service de comparaison car au delà d'une simple comparaison structurelle ce service fournit la liste d'actions à effectuer entre deux modèles.

FIGURE 6.5 – Extrait méta-modèle : nœud et nœud type



Ces actions sont dépendantes du nœud et ne sont pas définies de façon générique, de sorte que chacune peut redéfinir complètement ce qu'il faut effectuer pour passer d'un modèle à un autre.

On appelle *primitive d'adaptation* une action qui permet de faire passer une plate-forme d'un état A à un état B.

Déf 11. *AdaptationPrimitive(A,B)* $\equiv (A + AdaptationPrimitive(A, B)) = B$

Un modèle Kevoree représente un état instantané d'une plate-forme. Cet état est divisible en états intermédiaires et donc en *AdaptationPrimitive* intermédiaires, à l'image d'une forme currifiée [5]. La différence entre deux modèles peut alors s'exprimer à l'aide d'un ensemble de *AdaptationPrimitive* comme suit :

Déf 12. *AdaptationModel(M1 :KevModel,M2 :KevModel) ≡ Set<Primitive d'adaptation> à exécuter pour passer d'une plate-forme modélisée par M1 à une représentant M2.*

Si les modèles M1 et M2 doivent être valides structurellement, les états intermédiaires ne sont quant à eux, pas soumis à cette contrainte. Ce modèle d'adaptation est donc la donnée échangée en le M@R et le nœud instance.

Cette abstraction en *AdaptationPrimitive* assure une séparation forte des préoccupations, toujours par souci d'extensibilité. En effet les primitives d'adaptation suivent un patron de conception type/instance et sont alors typées. Ce type représente leur sémantique (AjoutComposant, SuppressionComposant, etc). La sémantique d'un nœud type se définit alors par les types de primitives d'adaptations qu'il peut émettre dans ses modèles d'adaptation à la suite d'une comparaison (*cf* 1, figure 6.4). Par exemple un nœud type Java va définir des primitives pour le chargement à chaud des fichiers JAR [6] et l'ajout de composants. Un nœud représentant une plate-forme embarquée (par exemple un micro-contrôleur AVR [Tur97]) va définir la primitive type d'ajout de composant mais non de chargement de type.

Le problème annoncé dans l'état de l'art concernant l'extensibilité des adaptations globalement prise en compte par le DDAS se ramène donc à l'extensibilité des types

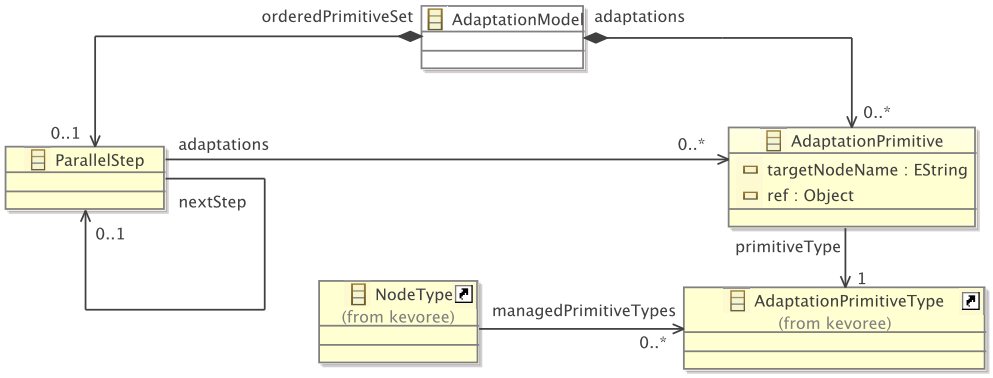
5. http ://en.wikipedia.org/wiki/Currying
6. http ://en.wikipedia.org/wiki/JAR_(file_format)

définis dans ses échanges. Dans le modèle Kevoree , les types de primitives d'adaptation sont portés par les *NodeType* via par la relation *managedPrimitiveTypes* de la figure 6.5. Chaque *NodeType* vient donc avec son bagage d'*AdaptationPrimitiveType* , le système global étend donc ses adaptations disponibles en fonction des types de nœuds que contient le modèle d'architecture.

6.2.2 Méta-modèle d'adaptation

Un méta-modèle supplémentaire est introduit pour définir le modèle d'adaptation entre deux modèles d'architecture Kevoree . Ce modèle d'adaptation capture la notion d'ensemble de primitives d'adaptation nécessaires pour le passage d'un état du DAS locale à un autre, ceci est illustré par la figure 6.6.

FIGURE 6.6 – Méta-modèle d'adaptation



Les *AdaptationPrimitive* étant fonction des modèles source et cible, ce nouveau méta-modèle est donc une décoration de deux modèles structurels Kevoree . Cette décoration se fait via les AdaptationPrimitive qui pointent vers les *AdaptationPrimitiveType* contenus dans les *NodeType* et possèdent une référence vers l'entité Kevoree impactée (par exemple un composant ou un *Channel*).

L'autre fonction de ce méta-modèle est d'encapsuler la notion de planification. En effet, ce besoin exprimé depuis les premiers processus MAPE vise à ordonner les actions afin de résoudre les conflits d'exécution entre elles. De façon schématique ce méta-modèle définit que les actions sont contenues dans des étapes. Ces étapes peuvent se chaîner les unes aux autres pour modéliser une séquence d'actions. Les actions au sein d'une même étape peuvent s'exécuter de façon parallèle. Pour planifier une exécution séquentielle d'un ensemble d'actions, il faut donc chaîner des étapes ne contenant qu'une seule action.

6.2.3 Responsabilités et fonctionnalités d'un nœud type

Le nœud instance est un élément passif qui reçoit des ordres du processus M@R sous forme de modèle à appliquer sur la plate-forme pilotée. Le processus du M@R qui

est plus détaillé en section 6.3 exploite les services du nœud instance pour générer la liste des primitives d'adaptation à appliquer localement pour passer d'un état A vers un état B. Cette comparaison se limite à la production des primitives d'adaptation qui concernent uniquement le nœud local dont le processus M@R a la responsabilité. Les adaptations qui concernent les autres nœuds sont à exécuter par eux-mêmes lorsque le modèle leur sera transmis, ceci afin de garantir l'encapsulation. Cette comparaison détaillée en sous-section 6.2.5 produit un résultat qui est dépendant des adaptations gérées par le nœud, ce résultat est donc l'interprétation en terme d'*AdaptationPrimitive* d'une différence entre deux états.

La génération du modèle d'adaptation n'est donc pas uniquement une différence structurelle entre deux modèles, cette dernière doit-être suivie d'une traduction en fonction de la sémantique d'adaptation locale et donc des *AdaptationPrimitiveType* contenus dans le *NodeType* . L'hébergement de l'algorithme de comparaison est de la responsabilité du *NodeType* car il correspond à son interprétation locale de la différence entre deux modèles. Cette comparaison doit assurer la garantie de migration de la plate-forme sous-jacente.

De plus, afin de pouvoir appliquer ces primitives d'adaptation abstraite, le *NodeType* a également la responsabilité de fournir un service d'équivalence entre les *PrimitiveAdaptation* et des commandes concrètes associées dont l'interface est définie en Java dans le listing 6.1 et suit le patron de conception Command [ERRJ95].

Listing 6.1– Interface des commandes concrètes d'adaptation

```
public interface PrimitiveCommand {
        public void execute() throws Exception;
        public void undo();
}
```

Les transitions d'état *via* l'application de ces commandes peuvent provoquer des erreurs qui doivent être remontées au processus de pilotage M@R. L'ensemble des commandes concrètes fournies par les nœuds types sont inversables pour permettre une annulation de la transition et revenir à l'état A. Ces commandes sont donc exploitables pour effectuer un mécanisme de *rollback* de récupération d'erreur lors d'une migration.

Pour conclure les nœuds types ont donc la responsabilité de fournir l'algorithme de comparaison de modèle spécifique à leur plate-forme ainsi qu'un canal inversable pour l'exécution. Leur contrat d'interface est illustré en Java dans la figure 6.2

Listing 6.2– Interface de nœud type

```
public interface NodeType {
        ...
        public AdaptationModel compare(Model current, Model target);
        public PrimitiveCommand getConcreteCommand(AdaptationPrimitive p);
}
```

6.2.4 Extensibilité des NodeType : héritage

À l'image d'un pilote de matériel les *NodeType* nécessitent des mécanismes de composition pour ne pas redéfinir l'ensemble de la sémantique d'adaptation à chaque défini-

tion. A l'inverse, bien qu'ils puissent être chacun indépendants, les *NodeType* partagent le plus souvent une sémantique d'adaptation commune qu'il faut capturer et réutiliser. Par exemple la plupart des *NodeType* définissent l'adaptation type *AjouterComposant*.

À l'image de l'héritage de classes en programmation par objets, les *NodeType* comme toutes les *TypeDefinition* peuvent définir une relation d'héritage entre eux. Cet héritage permet de récupérer et raffiner la sémantique d'adaptation d'un nœud parent en y ajoutant des notions d'adaptations. On définit alors qu'un héritage impose de respecter la notion de covariance et donc doit définir un ensemble plus grand que celui du parent. Cette covariance impose ici qu'un nœud fils définisse un ensemble de *AdaptationPrimitiveType* qui inclut au moins l'ensemble de *AdaptationPrimitiveType* définis par le parent. Celui est explicité par la définition suivante :

Déf 13. $N1 <: N2 \rightleftharpoons (N1.managedPrimitiveTypes \in N2.managedPrimitiveTypes)$

Pour raffiner la sémantique du *NodeType* parent les nouveaux sous-types peuvent exploiter un mécanisme de surcharge qui peut opérer à deux niveaux. Dans sa version la plus complexe la surcharge consiste à rajouter une *AdaptationPrimitiveType* et à étendre le processus *Kompare* . Cette version nécessite de raffiner le processus de comparaison ainsi que celui de planification pour émettre la nouvelle primitive dans les modèles d'adaptation produits.

Dans les cas où la sémantique de comparaison peut être gardée, le nouveau *NodeType* peut uniquement surcharger l'association des commandes concrètes pour définir le pilote d'une plate-forme différente mais manipulant les mêmes concepts que la plate-forme parent. Cette surcharge moins coûteuse permet d'organiser un ensemble de plates-formes cohérent sous un même type ancêtre qui devient une interface commune. C'est le cas par exemple avec les nœuds pour la plate-forme Java détaillée ci-dessous, qui partage la comparaison entre une plate-forme OSGi ou SCA.

6.2.5 Modèle d'adaptation et comparaison de modèle

Le processus M@R repose sur l'opérateur de comparaison qui permet de quantifier la différence entre un modèle représentant l'état courant du DDAS et un modèle représentant l'état cible.

La comparaison de modèles présentée dans cette thèse est une extension du processus présenté dans la thèse de Brice Morin, qui s'appuyait sur un opérateur d'intersection de modèle pour calculer les différences et produire un script d'adaptation en résultat. Dans l'approche de Morin un modèle noté *M1* était comparé avec un modèle cible *M2*, les éléments de l'ensemble (M2-M1) étaient ajoutés et ceux de (M1-M2) étaient supprimés, ceux issus de (M1∩M2) restaient inchangés.

Le processus de comparaison du modèle Kevoree , nommé *Kompare* étend l'approche de Morin *et al.* pour faire face aux nouveaux besoins dus à l'usage sur les DDAS. Cette approche cherche principalement à rendre extensible ce processus afin de faire face à la multiplication des types de plate-forme gérés mais également à prendre en compte la conception continue. Comme annoncé dans la section précédente, *Kompare* est raffiné par chaque *NodeType* pour faire face à ses spécificités. Son but est de fournir

en sortie un résultat de comparaison de deux modèles permettant au M@R d'exécuter les modifications de la plate-forme pour passer d'un état à l'autre.

6.2.5.1 Extension de l'opérateur d'intersection

La comparaison de modèle exploite l'opérateur d'intersection ∩ pour détecter les parties communes et donc stables des modèles. Cette similarité se complexifie avec l'usage de la conception continue, en effet des éléments similaires peuvent ne pas être stables et nécessiter une mise à jour si leurs binaires ont évolué. Par exemple un composant présent dans un modèle E1, et toujours présent dans un modèle E2 mais avec une version de binaire différente va être mis à jour si la plate-forme doit passer de l'état E1 à E2. Dans *Kompare* , l'opérateur d'intersection exploite donc la définition 3 (≡) pour détecter les parties communes des deux modèles et la définition 4 (=) pour émettre des primitives de mise à jour d'unité de déploiement et des instances associées par transitivité.

6.2.5.2 Planification

L'étape de planification cherche à optimiser les actions d'une adaptation afin de minimiser l'impact de son application sur le système. Par exemple si des composants doivent être mis en pause pendant l'adaptation, la planification cherche à minimiser ce temps d'arrêt de traitement. La planification se ramène donc à la recherche d'un chemin optimal d'exécution entre des actions reliées les unes avec les autres par des contraintes. Par exemple un composant a besoin d'être installé avant d'être démarré.

Ce problème a été largement abordé dans la littérature avec des approches très différentes :

- Approches généralistes avec le langage PDDL qui vise à offrir une abstraction de haut niveau pour la planification [FL03],[MGH+98]
- Approches par solveur de contraintes [JRL08]
- Approches spécifiques par parcours de graphe représentant les contraintes des actions [BF97]

Dans le modèle Kevoree les actions sont donc des *AdaptationPrimitive* à ordonner suivant les contraintes de déploiement. Les contraintes de déploiement étant spécifiques à chaque plate-forme, ce traitement est du ressort du *NodeType* . Cette étape est incluse dans la définition du *NodeType* à la fin du service de comparaison. Les contraintes de chaque plate-forme sont différentes, ainsi que la planification qui en résulte. L'approche Kevoree ne contraint en rien le système de planification à utiliser. Cependant la capacité de Kevoree à spécifier les *NodeType* pousse pour des raisons de performance à exploiter des approches spécifiques. Le reste de la sous-section est donc illustré avec la planification exploitée pour le *NodeType* Java qui est suffisamment générique pour illustrer ce mécanisme. Dans ce cas une résolution par parcours de graphe est exploitée, qui s'avère suffisamment expressive vis-à-vis des contraintes.

Premièrement il est nécessaire d'exprimer les contraintes entre les actions. En voici un extrait pour le nœud Java :

- un composant dépend d'une l'installation de son type ;

- un composant dépend d'un *Channel* s'il est connecté *via* un de ses ports requis ;
- un *Channel* dépend d'un composant s'il est connecté *via* un de ses ports offert ;
- une mise à jour d'un composant dépend de la mise à jour de son binaire ;
- etc.

Il est alors possible de construire un graphe orienté dont les nœuds sont les actions et dont les contraintes sont les arcs. Si le graphe n'est pas un graphe sans circuit alors la planification est abandonnée puisque cela veut dire que deux actions dépendent l'une de l'autre, par exemple si un composant dépend d'un autre pour son démarrage et vice-versa. Dans ce cas la planification est rejetée et le modèle est rejeté puisque non exécutable sans risque d'inter-blocage. L'API Java des composants Kevoree permet de déclarer ou non ces contraintes de démarrage pour limiter ces risques de cycle. Un algorithme de tri topologique [7] est ensuite appliqué pour rechercher un ordre de parcours optimal. Cet algorithme est implanté avec une approche gloutonne [8] afin de séparer les étapes de parcours (recherche de nœuds sans arcs entrants). Ces étapes de parcours permettent de détecter les étapes exécutables en parallèle. Ces étapes donnent lieu à une création d'une *Step* incluant ces tâches dans l'*AdaptationModel* résultant de la planification. Le détail de l'algorithme de planification sort largement du sujet de cette thèse, il est cependant nécessaire de garder à l'esprit que cette étape est essentielle pour l'exécution des adaptations.

Distribution de la plannification ? La question de la distribution ou non de la planification (au sens ordonnancement des actions) des adaptations est difficile. Ce choix est directement lié à la centralisation ou non de la prise de décision de l'adaptation et même du contrôle de l'application de celle-ci.

Dans les cas où le calcul de l'adaptation est centralisé, l'approche la plus commune est de centraliser également la planification des adaptations. Ainsi dans le domaine des grilles de calcul [ABP+03] [ABP05], la planification d'adaptation devant s'exécuter sur plusieurs machines a été envisagée. En d'autres termes, un seul noeud calcule alors l'ensemble des actions à effectuer sur les différents noeuds et détermine un ordre d'exécution résolvant les conflits et dépendance entre ces actions. Dans ces cas d'usage grille, l'ordonnanceur peut alors surveiller l'exécution de l'adaptation sur chaque noeud depuis un noeud central.

Cette centralisation pose problème dans le cas de DDAS mobile et dont la connectivité est sporadique et surtout dans les cas où la prise de décision est elle même distribuée. A l'inverse le modèle Kevoree préconise une responsabilité d'adaptation de chaque nœud et une isolation des processus M@R. Cette isolation permet la divergence du DDAS et ainsi des mises à jour sur des topologies P2P. L'exécution distante d'adaptation rentre en conflit avec ce principe d'isolation et responsabilité des nœuds.

Les groupes permettent déjà de faire du consensus et ainsi de faire de la collaboration sur la décision de migration vers un nouveau modèle, ceci correspond à de la planification gros grain. L'expression à un grain plus fin (au niveau des exécutions de

7. http ://en.wikipedia.org/wiki/Topological_sorting
8. http ://fr.wikipedia.org/wiki/Algorithme_glouton

chaque commande) peut être nécessaire dans le cas de liaisons entre composants ne pouvant pas faire de rétention en cas d'indisponibilité du réseau par exemple pendant le déploiement sur le nœud distant. De fait la planification grain fin est nécessaire dans des cas de contraintes entre démarrage de composant distribué, ou encore dans des cas de migration d'état entre noeuds qui nécessite par exemple qu'un composant soit arrêté d'une plate-forme avant d'être redémarré sur une autre.

Pour les cas d'usage mobiles, la solution de contournement à ce problème consiste à utiliser des *Channel* et *Groupe* qui sont sujets à fragmentation et donc à des possibilités d'attente de déploiement distants et sont capables de faire de la rétention et du rejeu pour résister à des pannes réseaux entre fragments. Pour les autres cas d'usage il est nécessaire de coordonner les différentes boucles MAPE des noeuds. Les patrons de contrôle décentralisé de Weyns *et al* [WSG+12] apportent plusieurs réponses à ce problème. Dans l'application de ces patrons chaque instance de MAPE doit permettre de renvoyer son état d'exécution courant et d'attendre un état d'un noeud voisin.

Pour permettre l'implantation de ces deux primitives Kevoree propose deux mécanismes. D'un coté chaque exécution d'une commande donne lieu à l'émission d'un évènement du core vers les groupes. Les groupes peuvent alors décider ou non de disséminer ces évènements à leurs fragments distants. D'un autre coté la planification peut déterminer qu'une étape d'adaptation locale nécessite l'attente d'un message distant et permet ainsi de faire des points d'arrêt dans l'adaptation, et pour cela un type spécifique de *Step* est utilisé. Ce mécanisme permet la construction de cycle MAPE collaboratif et ainsi permet la planification distribuée sans briser l'encapsulation des noeuds vise à vis de leur processus MAPE. Cependant ceci repose sur une assomption forte, les noeuds et les groupes doivent assurer de l'homogénéité de la planification sur le DDAS sans quoi l'émission des messages ou les étapes d'attente ne seraient pas cohérentes et mettraient en péril l'application de l'adaptation. La coordination d'un tel mécanisme est complexe et ceci explique que beaucoup d'approches centralisent cet ordonnanceur dans les cas de grille de calcul.

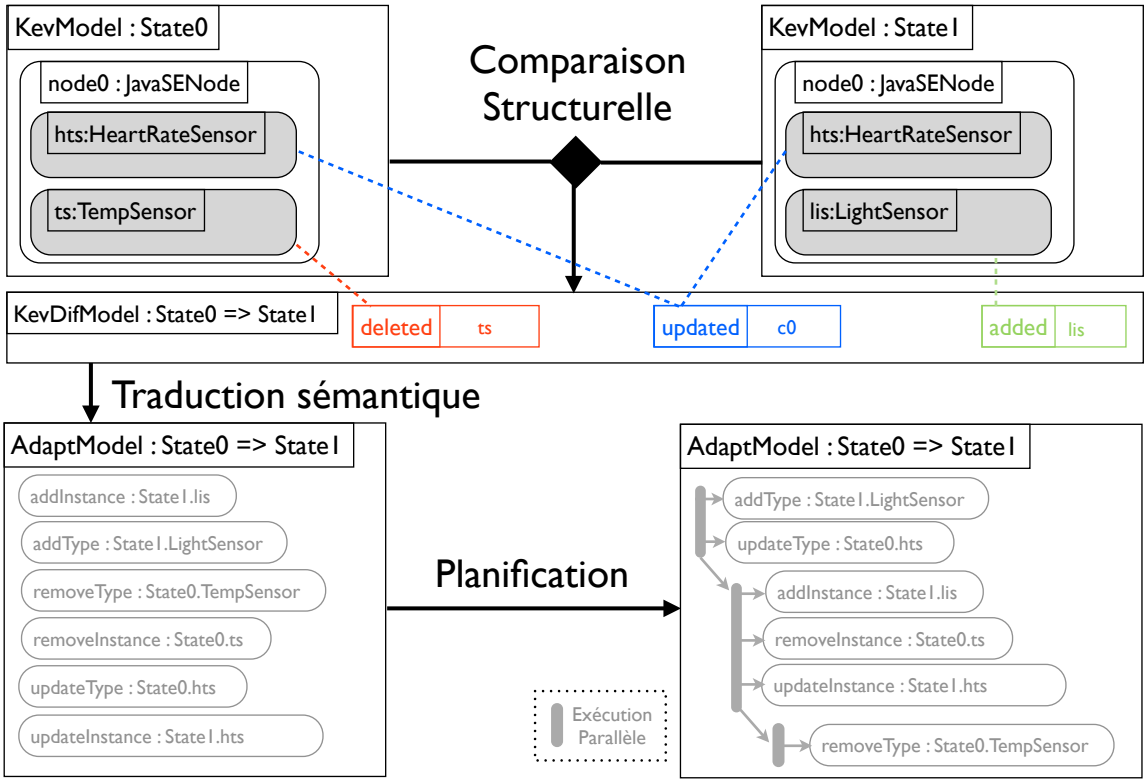
La description précise de ce mécanisme sort du cadre de cette thèse, car il est difficilement applicable dans le cas des réseaux DDAS sporadiques mobiles qui sont la principale motivation du modèle Kevoree . La planification distribuée bien que réalisable pose des assomptions difficilement conciliables avec les besoins de systèmes dynamiques ouverts, ce qui en limite les cas d'usages possibles.

6.2.5.3 Détail du processus Kompare

Le processus *Kompare* correspond à l'assemblage des trois étapes : comparaison structurelle, traduction sémantique et enfin planification des *AdaptationPrimitive* . La figure 6.7 illustre ce processus sur une adaptation qui comprend une suppression, mise à jour et un ajout de composant.

La comparaison structurelle exploite donc uniquement les opérateurs décrits précédemment. La traduction sémantique est spécifique au noeud (en l'occurrence *Java*), de manière pragmatique il s'agit de mettre en corrélation des commandes concrètes en face d'un modèle abstrait. Il en ressort un *AdaptModel* non planifié. La planification

FIGURE 6.7 – Structuration du processus Kompare



rajoute des étapes qui permettent d'organiser les commandes en exécution parallèle ou séquentielle. Les commandes sont alors organisées sous la forme d'un arbre, prêtes à être exécutées.

6.2.6 *Mapping* vers plates-formes concrètes

Les concepts d'adaptation du modèle Kevoree ont été implantés pour gérer des plates-formes très différentes. Cette diversité des implantations fait partie d'un axe de validation, cette section survole donc uniquement ces implantations détaillées par la suite dans la partie validation. D'un côté une version Java permet de faire de l'adaptation sur des environnements puissants et portables, et d'un autre une implantation pour les micro-contrôleurs AVR permet de faire de l'adaptation sur des environnements très contraints. La version Java a été par la suite étendue pour fonctionner dans des environnements faiblement contraints tels que la plate-forme Android. Dans un autre axe, des travaux ont été menés conjointement avec la thèse d'un autre doctorant pour adapter ce concept d'adaptation sur des environnements d'infrastructure de type *Cloud computing*. Cette adaptation est abordée dans la section consacrée aux perspectives.

6.2.6.1 Classification des niveaux d'adaptation

Les *NodeType* sont donc caractérisés par leurs *AdaptationPrimitiveType* . Celles-ci peuvent être classifiées pour déterminer les niveaux d'adaptation que les nœuds instances seront capables de fournir. Dans ses capacités d'adaptation, on distingue les niveaux suivants :

- **Niveau 1 : Adaptation paramétrique** Cette classe identifie les mises à jour des paramètres des *Instance* . En effet chaque clé d'un dictionnaire de port peut être modifiée à chaud, par exemple pour changer la valeur d'échantillonnage d'un capteur physique ou encore le port d'un serveur Web.
- **Niveau 2 : Adaptation architecturale** Identifie les modifications du graphe d'*Instance* de Kevoree. Ainsi l'ajout, la suppression dynamique d'un composant, *Channel* , nœud, par exemple pour élargir un *cluster* de serveur pour répondre à une plus forte charge.
- **Niveau 3 : Chargement à chaud de *TypeDefinition*** Toutes modifications du graphe des *TypeDefinition* et des *DeployUnit* après le chargement initial. Cette classe de reconfiguration permet de modifier à chaud le comportement d'une instance et donc d'assurer une conception continue.
- **Niveau 4 : Adaptation de nœuds distants** Les nœuds supportant cette catégorie participent à la gestion d'adaptation d'autres nœuds. En d'autres termes ils supervisent d'autres nœuds de plus faible puissance, ne pouvant pas héberger l'intégralité d'M@R. Cette catégorie correspond à une externalisation des services du M@R tout en garantissant l'encapsulation des nœuds nécessaires pour Kevoree .

Le tableau 6.1 classifie les *NodeType* suivant leurs niveaux d'adaptation gérés. Ce tableau est loin d'être exhaustif mais illustre les éléments les plus pertinents.

TABLE 6.1 – Classification des *NodeType*

Niveau d'adaptation	JavaSE	Dalvik Android	Arduino
Niveau 1	+	+	+
Niveau 2	+	+	+
Niveau 3	+	+ contraintes de performances	+/- flash complet de la mémoire
Niveau 4	+	+/- consommation batterie	- ressources insuffisantes

Ainsi on peut considérer que les nœuds Java de l'édition standard sont capables de gérer tous les niveaux d'adaptation. Les environnements faiblement contraints tels qu'Android sont également capables de mettre en œuvre tous les niveaux mais avec des contraintes dues à la mobilité. Ainsi le niveau 3 consomme de la bande passante tandis que le niveau 4 consomme de la batterie pour la gestion distante et donc l'établissement de connexion réseau. Enfin l'environnement Arduino (micro-contrôleur AVR 8 bits) est quant à lui capable d'adaptation de niveau 1 et 2. La conception continue est possible

mais coûteuse et les ressources sont bien évidement insuffisantes pour la gestion d'autres nœuds.

6.2.7 Modèle de topologie

Le modèle de topologie est un besoin exprimé dans l'état de l'art pour les systèmes distribués et notamment pour des organisations réseau pair à pair. Ces informations visent à synthétiser les données sur l'organisation réseau du DDAS et notamment les liaisons entre nœuds. Ces liaisons portent un certain nombre d'informations, sur leurs types, leurs vitesses de transfert ou simplement leurs états de panne ou non. Le modèle de topologie doit par exemple permettre de répondre à la question : sur quelle adresse IP un nœud est-il joignable ?

Comme dans les systèmes autonomes et pervasifs, ce modèle n'est pas statique mais à l'inverse s'enrichit durant le cycle de vie de chaque nœud de l'ensemble des informations recueillies. De la même manière que les informations structurelles des plates-formes telles que les nœuds et composants, les informations liées à la topologie nécessitent d'être partagées pour être regroupées et exploitées de manière individuelle sur chaque nœud. Ainsi par exemple dans un cas d'usage où un raisonneur cherche a optimiser la distribution des miroirs d'un service, ce dernier a besoin de l'ensemble des informations collectées par ses nœuds afin de répartir les miroirs en fonction des liaisons réseau. Pour laisser plusieurs nœuds prendre de telles décisions et ainsi améliorer la tolérance aux fautes, il est nécessaire de distribuer cette information et, de la même manière que le *Model@Runtime* , permettre au nœud d'exploiter ces informations *offline*.

6.2.7.1 Lier les informations topologiques au modèle structurel

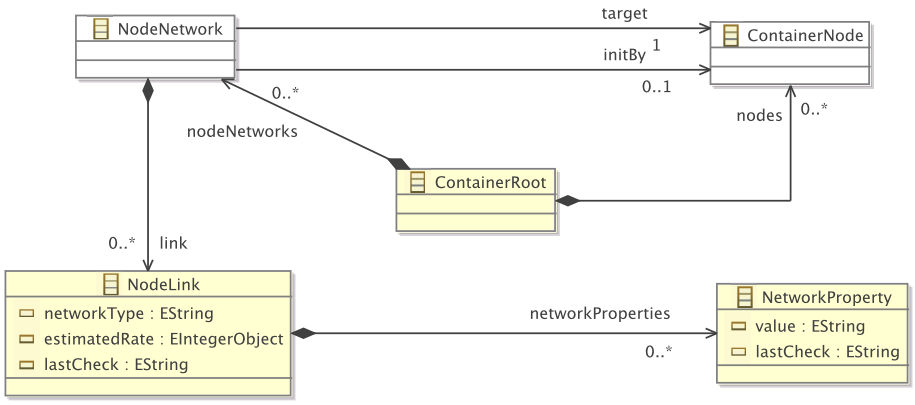
Le cycle de vie du modèle de topologie est donc très proche du modèle structurel exploité pour le *Model@Runtime*. Pour cette raison, ces deux informations sont regroupées dans le même modèle, qui suit alors les canaux de diffusion précédemment définis. Le modèle de topologie Kevoree se base donc sur le modèle de nœud qu'il vient enrichir. Un extrait du méta-modèle est présenté en figure 6.8.

Les informations topologiques s'y organisent sous forme d'un graphe orienté. Les nœuds de ce graphe sont les nœuds Kevoree tandis que les arcs sont représentés par les *NodeLink* . Ces liens représentent une information orientée puisque issue d'un *initNode* vers un *targetNode* . Un nœud A peut représenter sa réussite à joindre un nœud B en rajoutant un lien avec A comme *initNode* et B comme *targetNode* .

Les *nodeLink* ont également un attribut qui définit leurs types (par exemple Wifi, Ethernet, ou Radio) ainsi qu'une estimation du débit de transfert. Cette information permet aux raisonneurs de connaître par exemple l'état de charge du DDAS d'un point de vue échange réseau. En supplément, les liens du graphe peuvent porter des informations supplémentaires *via* un ensemble de *NetworkProperty* , ceci permet de garder le modèle ouvert à des informations nécessaires dans des cas uniques, comme par exemple un SSID Wifi.

Enfin l'ensemble des informations des liens ont un *tag* nommé *lastCheck* qui permet

FIGURE 6.8 – Modèle de topologie des nœuds



de connaître la dernière date de vérification et ainsi prendre en compte le caractère volatile des topologies réseaux prises en compte par cette thèse.

6.3 Model@Runtime Core

Le *Model@Runtime Core* est le chef d'orchestre de l'adaptation dynamique dirigée par les modèles. C'est également le point d'entrée qui permet d'interagir avec les primitives du processus MAPE. Ce processus, qui s'apparente à un automate, est responsable d'un nœud dirigé par le *Model@Runtime*, et une copie de ce processus est déployé sur chaque nœud. Comme cela a été annoncé au début de la contribution, la construction d'un DDAS dirigé par le *Model@Runtime* requiert donc l'assemblage et la synchronisation d'autant de processus que de nœuds.

En contre-partie de cette duplication, chaque nœud hérite donc d'une certaine indépendance de son adaptation locale via-à-vis de celle du DDAS. En d'autres termes chaque processus peut s'adapter localement sans appel nécessaire à un superviseur. Ce type d'architecture vise donc à répondre au besoin d'architectures capables de continuer à s'adapter dans des réseaux non fiables et donc dans le cas des DDAS ayant perdu des nœuds.

Si ce processus permet à chaque processus *Model@Runtime* de diverger, il faut assurer les moyens de resynchronisation. L'objectif de cette section est de couvrir la définition de ce processus ainsi que ses possibilités d'intercepteurs et verrous destinés à la synchronisation.

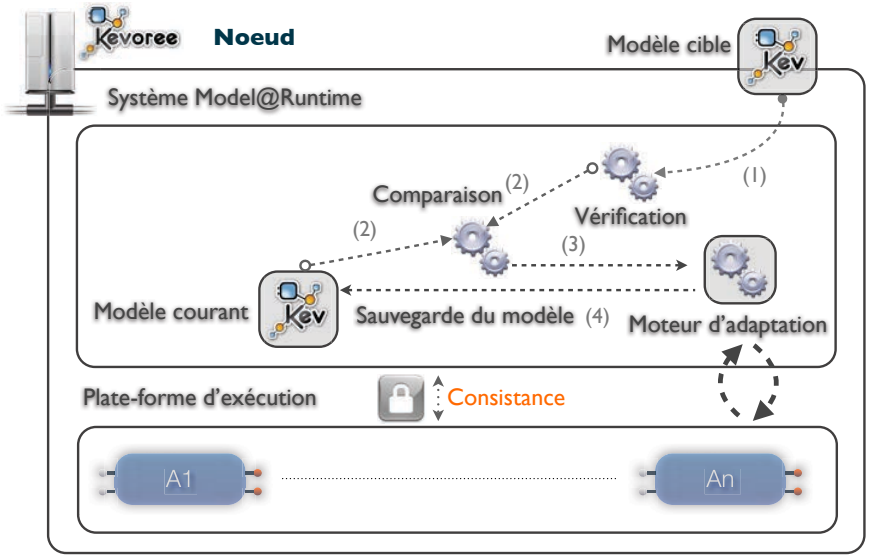
6.3.1 Définition

L'objectif du Model@Runtime Core (M@RC) est d'assurer une synchronisation forte entre la plate-forme et une version du modèle d'architecture Kevoree appelé modèle courant. Pour cela il est déclenché sur demande avec une proposition d'un nouveau

modèle, et il assure la synchronisation en exploitant les services précédemment définis d'un nœud instance afin de piloter la plate-forme.

Le M@RC doit assurer une encapsulation de son modèle courant, aucun autre processus ne peut le modifier sans passer par une proposition déclenchant l'ensemble du processus. Le M@RC doit également avoir un accès exclusif, non parallélisable sur la plate-forme pour assurer la cohérence de son état vis-à-vis du modèle. Le principal intérêt du *Model@Runtime* est de pouvoir effectuer un certain nombre de vérifications avant déploiement sur la plate-forme. Ainsi ce processus peut exploiter des *checkers*, qui sont des processus qui valident ou non la cohérence du nouveau modèle proposé. La figure 6.9 donne une première illustration de ce processus.

FIGURE 6.9 – Processus Model@Runtime Core



De façon schématique le processus fait donc une comparaison puis un calcul de modèle d'adaptation grâce au nœud instance puis une application avant de sauver le nouveau modèle comme état courant puisque synchronisé. Le listing 6.3 illustre ceci par une implantation dans le langage Scala.

Listing 6.3– Script Model@Runtime Core

```
def mar_core_proces(node:NodeInstance,newmodel:KevModel) : Boolean = {
        if(!check(newmodel)) {throw error}
        val compare_result : AdaptationModel = node.compare(currentModel,newModel)
        if(!executeAdaptation(compare_result)){throw error}
        keepHistory(currentModel)
        setCurrentModel(newModel)
}
```

En cas d'erreur ou si le modèle est détecté comme non conforme le modèle est simplement rejeté. L'algorithme d'exécution d'adaptation (méthode *executeAdaptation*) a la charge d'inverser les primitives déjà appliquées pour retomber sur un état corres-

pondant à l'état courant. Le modèle courant est également gardé dans un historique avant d'être remplacé, afin de garder une trace des états précédents. Grâce à ce mécanisme il est possible de détecter le problème de *state flapping* c'est-à-dire un système qui oscille autour de plusieurs états très proches et boucle sur une succession de mises à jour. La détection du *state flapping* est un problème difficile et encore ouvert, le but du *Model@Runtime* est de fournir *via* son historique un support pour sa détection.

6.3.2 Exécution de modèle d'adaptation et reprise sur erreur

Le moteur d'exécution des modèles d'adaptation a pour but d'appliquer les primitives d'adaptation suivant l'ordre défini par les étapes contenues dans celui-ci. Ces étapes sont le résultat de la planification effectuée par les processus *Kompare* du nœud instance. L'interpréte traverse donc ces étapes tant que l'exécution des commandes concrètes ne renvoie pas d'erreur. En cas d'erreur toutes les adaptations déjà effectuées sont inversées (*cf.* l'interface primitive *commande*) dans l'ordre inverse du modèle d'adaptation et une erreur est retournée au *Core* . Ce mécanisme permet de garder la cohérence de la plateforme même en cas d'erreur. En effet si la plupart des erreurs doivent être détectées par les *checker*, les erreurs d'exécution peuvent survenir pour des problèmes de contexte d'exécution (par exemple débordement mémoire, disque, etc). Le listing 6.4 représente l'implantation de ce processus en Scala. Le point d'entrée est donc la méthode *"exec"* qui prend en paramètre le noeud courant ainsi que le modèle d'adaptation. Pour chacune des *AdaptationPrimitive* , l'algorithme cherche alors la commande concrète associée et l'exécute. Dans le cas d'une remontée d'exception lors de cette exécution, la méthode *"rollback"* est appelée, dans le cas contraire l'interprétation de la prochaine *Step* est appelée.

Listing 6.4– Adaptation Execution Engine

```
def exec(node:NodeInstance,adaptationModel:AdaptationModel) : Boolean = {
        exec_step(adaptationModel.getOrderedPrimitiveSet()))
}
def exec_step(node:NodeInstance,step : ParralelStep) : Boolean = {
        var primitives = List[PrimitiveCommand]
        val execResult = step.getAdaptations.parralel.foreach { adapt =>
                val primitive = nodeInstance.getPrimitive(adapt)
                primitives += primitive
                primitive.execute
        }
        if(execResult){
                if(step.getNextStep() != null){
                        if(exec_step(node,step.getNextStep())){
                                return true
                        } else {
                                rolback(primitives);false
                        }
                } else {
                        return true;
                }
        } else {
                rolback(primitives);false
        }
}
```

```
def rollback(primitives : List[PrimitiveCommand]){
        primitives.foreach{ primitive =>
                        primitive.undo()
        }
}
```

L'implantation du moteur d'exécution est donc un interprète récursif. Les éléments importants à noter ici sont l'exécution en parallèle des primitives d'une même *step* mais surtout le déclenchement du processus de *rollback* en cas d'échec de l'une des commandes du même niveau ou du niveau inférieur.

6.3.3 Extensibilité et interruptilibilité du processus Core

De manière uniforme les algorithmes de synchronisation de données ou processus abordés dans l'état de l'art exploitent des primitives d'interruption afin de déclencher les échanges de synchronisation entre nœuds distribués. Ces primitives peuvent être de plusieurs natures : interception bloquante de processus, pose de verrous ou encore modèle évènementiel asynchrone. Cette diversité s'observe également dans les étapes de mise à jour où ces processus interviennent. Ainsi de manière non exhaustive on observe les cas suivants :

- un processus de type *Paxos* ou consensus devra s'inscrire en interruption bloquante avant la mise à jour locale dans le but de demander un consensus avec les autres nœuds sur cette valeur avant de la mettre à jour ;
- un processus de type *gossip* s'inscrira en interruption non bloquante après la mise à jour locale dans le but de propager l'information de mise à jour locale ;
- un processus de type *broadcast* peut également s'inscrire de manière bloquante après la mise à jour locale pour attendre un acquittement et dans le cas contraire déclencher un phase de *rollback*.

Ces pré et post interruptions peuvent donc être exploitées de manière très diverses.

Par extrapolation, le processus M@RC nouvellement défini fait partie des processus à synchroniser par de tels algorithmes pour faire collaborer plusieurs nœuds et construire un DDAS. Il doit donc fournir ces primitives qui encadrent les appels de mise à jour de la valeur locale, qui est ici le modèle courant. Le M@RC doit fournir des interruptions *Pre* et *Post* de mise à jour de modèle, bloquante ou non. Comme annoncé, les mécanismes de synchronisation dans le modèle Kevoree doivent être dynamiquement adaptables, ainsi les algorithmes doivent pouvoir dynamiquement s'inscrire en étendant du même coup le processus M@RC.

Afin de répondre à ces deux besoins, la notion de *ModelListener* est introduite, ainsi qu'une notion de verrou détaillée en sous-section 6.3.4.

Déf 14. *ModelListener ≡ Entité encapsulant un mécanisme de synchronisation et dédiée à l'extension du processus M@RC. Appelées de façon séquentielle, ces entités doivent s'acquitter de leur synchronisation avant et après la mise à jour de modèle.*

Le listing 6.5 définit en Scala l'interface de telles entités.

Listing 6.5– Interface de ModelListener

```
def preUpdate(ContainerRoot currentModel, ContainerRoot proposedModel) : Boolean
def preAllUpdate(ContainerRoot currentModel, ContainerRoot proposedModel) : Boolean
def postUpdate(ContainerRoot currentModel, ContainerRoot proposedModel) : Boolean
def postAllUpdate(ContainerRoot currentModel, ContainerRoot proposedModel) : Boolean
def asyncAfterUpdate(ContainerRoot newCurrentModel)
def preRollBack(ContainerRoot currentModel, ContainerRoot proposedModel)
def postRollBack(ContainerRoot currentModel, ContainerRoot proposedModel)
```

La méthode *preUpdate* est appelée après l'étape de contrôle local et avant le déclenchement du processus de *Kompare*. La méthode *postUpdate* est appelée après l'exécution de moteur d'adaptation et avant la sauvegarde du modèle proposé en tant que modèle courant. Les phases de pré et post-traitement se font en deux passes, une première où les algorithmes évaluent la validité de la mise à jour, et la deuxième où la synchronisation est réellement effectuée. Cela contribue à la résolution du problème de composition qui se pose immédiatement après avoir permis leurs extensions dynamiques. Ce point est discuté plus en détails ci-dessous. Les méthodes *preAllUpdate* et *postAllUpdate* sont donc appelées respectivement quand l'ensemble des *pre* et *post update* ont répondu *true*. En cas de retour négatif lors des méthodes du *ModelListener* , le modèle est rejeté et les modifications engagées annulées. Les méthodes *preRollBack* et *postRollBack* des *ModelListener* sont respectivement appelées avant et après cette phase de retour afin de faire les traitements de nettoyage des commandes si nécessaire. Le listing 6.6 explicite l'extension de l'implantation nécessaire dans M@RC pour mettre en œuvre cette fonctionnalité.

Listing 6.6– Script étendu du Model@Runtime Core

```
def mar_core_proces(node:NodeInstance,newmodel:KevModel,
  listeners:List[ModelListener]) {
        if(!check(newmodel)) {throw error}
        if(!preCall(newmodel,listeners)) {throw error}
        val compare_result : AdaptationModel = node.compare(currentModel,newModel)
        if(!executeAdaptation(compare_result)){throw error}
        if(!postCall(newmodel,listeners)) {rollback(compare_result);throw error}
        keepHistory(currentModel)
        setCurrentModel(newModel)
        listeners.foreach{l=>l.asyncAfterUpdate(listeners)}
}
def preCall(newmodel:KevModel,listeners:List[ModelListener]):Boolean={
        sort(listeners).forall(l=>l.preUpdate(currentModel,newmodel))
       && sort(listeners).forall(l=>l.preAllUpdate(currentModel,newmodel))
}
[...] idem pour postCall [...]
```

6.3.3.1 Difficulté de la notion d'ordre et de composition des ModelListeners

La méthode *sort()* du listing précédent illustre le problème difficile qu'est la composition et l'ordonnancement des *ModelListeners* . En effet la relation d'ordre d'appel des *ModelListeners* peut être critique, notamment lorsque certains ont des effets de bords. Par exemple un algorithme peut nécessiter que le *ModelListener* soit appelé en dernier ou premier. Cela est dû principalement à la nature très différente des *ModelListeners* ; certains sont par exemple des extensions de l'étape de contrôle, tandis que d'autres effectuent des communications. La séparation en deux passes des étapes de *pre* et *post* est

déjà une première réponse à ce problème ; en effet les contrôles (*checks*) s'inscrivent à la première passe et la communication se réalise dans la deuxième. De même ce mécanisme autorise des algorithmes de pré-réservation dans la première passe.

Pour tous les autres cas de figure les *ModelListeners* définissent un niveau de priorité qui se retrouve alors interprété lors du classement de ceux-ci avant les appels de méthode. Cette réponse à la composition n'est que partielle et elle demanderait une approche par contrainte afin d'assurer la cohérence de la synchronisation. Par exemple un *ModelListeners* pourrait faire les demandes de contraintes suivantes : être exécuté avant un autre, être exécuté en dernier, premier, etc.. Ce point sera abordé dans la section validation de ce document.

6.3.4 Model@Runtime service : Api Mape

Le M@RC est le moteur d'exécution du *Model@Runtime* mais ce dernier n'est pas exposé de façon directe aux différents raisonneurs de modèle. Une façade de service est introduite au-dessus du M@RC pour fournir un accès à une partie du processus MAPE : le *monitoring* et l'exécution. L'analyse est en effet du ressort des dits raisonneurs et la planification est du ressort du nœud comme cela a déjà été abordé.

Ce service, qui est disponible pour toutes les instances locales d'un nœud, porte sur l'accès à l'exécution de changement d'état du système *via* des propositions de nouveaux modèles. Le *monitoring* est essentiellement géré par les *ModelListeners* , ce service ne fait alors que proposer l'accès aux abonnements.

Le listing 6.7 illustre l'implantation de ce service en Scala.

Listing 6.7– Interface du service Model@Runtime

```
trait ModelRuntimeService {
        //execution
        def (async)updateModel(model:KevModel)
        def (async)compareAndSwap(model:KevModel)
        //monitoring
        def (un)registerModelListener(ModelListener listener)
        def getCurrentModel() : KevModel
        def getPreviousModel(id:Long):KevModel
        //lock
        def acquireLock(timeout:Long) : LockID
        def releaseLock(id:LockID)
}
```

Les méthodes dédiées au *monitoring* permettent d'accéder au modèle courant (*getCurrentModel*) , à l'historique des version précédentes (*getPreviousModel*) et aux abonnements de *ModelListeners* (*(un)registerModelListener*).

6.3.4.1 Accès concurrent local au service Model@Runtime

Si la notion de groupe détaillée ci-dessous traite de la synchronisation inter plate-forme, la position centrale du *Model@Runtime* au sein d'un nœud force le M@RC à traiter les accès concurrents sur les demandes d'exécution. Les accès sont concurrents au sein même d'une plate-forme en raison de la multiplicité des algorithmes de groupes mais aussi de celle des modèles raisonneurs.

Ce problème se ramène à celui des accès concurrents à une zone de mémoire partagée, classiquement abordé en informatique et plus particulièrement au niveau des processeurs. On trouve dans la littérature deux grandes familles de solutions à ce problème : les approches à verrous et les approches *lock-free* non bloquantes.

L'approche par verrous [9] consiste à demander l'accès exclusif à une ressource pour un processus donné. Cet accès exclusif une fois obtenu, le processus peut effectuer ses traitements sans risque et peut alors libérer le verrou pour libérer les autres processus nécessitant un accès à cette ressource. Pendant qu'un verrou est posé tous les autres accès à la zone mémoire sont bloquants. Le *Model@Runtime service* offre donc un tel mécanisme pour protéger son accès *via* les méthodes *acquireLock* et *releaseLock* .

L'opérateur *compare-and-swap* [HFP02] permet la construction d'algorithme *lock-free*. Son principe est le suivant : au moment de mettre à jour une valeur, l'opérateur exige en même temps que la nouvelle valeur, l'ancienne valeur connue par le processus. Si l'ancienne valeur correspond à la valeur courante, l'opération échange alors les valeurs. Dans le cas contraire l'échange est annulé et une erreur peut être remontée. Ceci permet à des processus de travailler sur une valeur sans bloquer les autres et au moment de la modification cela permet de vérifier que la modification est toujours opportune. Ce mécanisme est donc offert dans la méthode *compareAndSwap* , la valeur à comparer est alors le modèle courant. Les modèles Kevoree sont comparables par hachage. Ainsi la méthode *compareAndSwap* permet à un raisonneur de travailler sur une mise à jour sur un modèle courant sans bloquer les autres, puis de proposer cette mise à jour au cœur en donnant la version initiale de sa modification. En cas de non égalité, si un processus a effectué une mise à jour, le modèle est rejeté et le processus doit recalculer sa modification à nouveau sur le modèle courant.

A l'aide de ces deux mécanismes, le *Model@Runtime* peut coordonner plusieurs accès concurrents, la classification et l'usage permettant de choisir l'une ou l'autre méthode est abordée par la suite.

6.4 Groupes de synchronisation

6.4.1 Encapsulation des sémantiques de synchronisation

Le modèle Kevoree définit une entité dédiée pour expliciter les sémantiques de synchronisation des nœuds. De cette façon, il est possible de passer d'une modélisation d'un DAS à celle d'un DDAS. De façon très simpliste, les groupes définissent des canaux de communication dédiés au transfert des évolutions d'états du DDAS et donc des modèles. Ces échanges doivent faire converger un ensemble de nœuds vers un état et donc un modèle commun. Il est alors à noter que dans cette approche les groupes sont définis dans un modèle dont ils ont eux-mêmes la responsabilité de diffusion.

Un groupe est caractérisé par la nature (synchrone ou asynchrone) des échanges générés mais surtout par le moment de déclenchement de ceux-ci. Pour cette raison, les groupes sont nativement des *ModelListener* tels que précédemment définis. En ef-

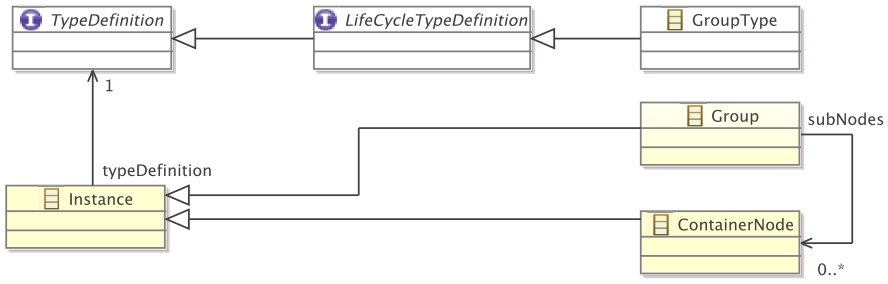
9. http ://en.wikipedia.org/wiki/Lock_(computer_science)

fet chaque sémantique de synchronisation doit étendre le M@RC pour intercepter les moments opportuns pour déclencher les échanges d'états.

Au-delà de simples *ModelListener* , les groupes définissent également une relation entre un ensemble de nœuds dont on attend une cohérence de l'état. De façon plus précise, l'assemblage des groupes permet d'assurer les points de cohérence entre les visions des modèles.

Comme les autres entités Kevoree , les groupes suivent le patron type/instance. Si un groupe assure une sémantique de synchronisation entre un ensemble de nœuds fils, l'algorithme distribué exploité pour les échanges est lui défini dans un *GroupType* . La figure 6.10 représente l'extrait correspondant du méta-modèle Kevoree .

FIGURE 6.10 – Extrait méta-modèle : groupe



Comme les autres instances, les groupes sont déployables à chaud entre plusieurs nœuds. Plusieurs sémantiques et donc plusieurs *GroupType* peuvent être intégrés dans un modèle Kevoree afin de représenter la diversité des algorithmes de convergence étudiés dans l'état de l'art. Les groupes illustrés dans la figure 6.10 définissent une relation *subNodes* vers des nœuds fils du point de synchronisation.

6.4.2 Fragmentation des groupes

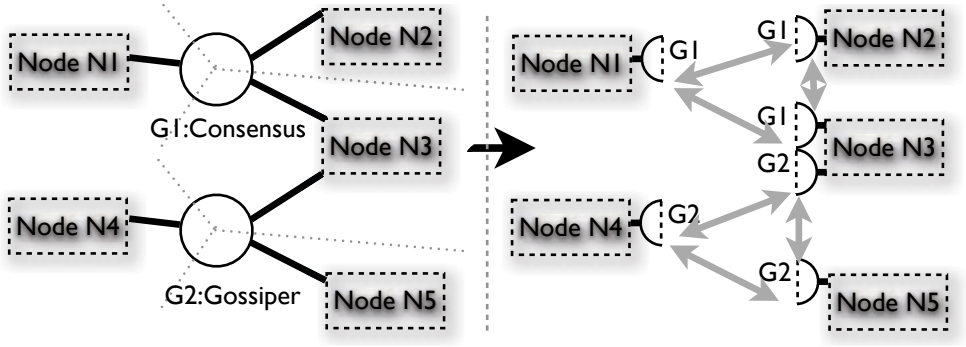
De manière analogue aux *Channels* les groupes sont des entités non localisées. Ils sont donc fragmentés sur les nœuds abonnés, et un agent de ce groupe est déployé sur chaque nœud abonné pour collaborer avec les autres. La figure 6.11 illustre ce mécanisme de fragmentation. Dans l'exemple le nœud N3, qui est abonné à deux groupes différents (G1 et G2), se retrouve avec deux fragments locaux qui vont chacun s'inscrire au M@RC.

6.4.3 Problème du Hara-Kiri

De façon inhérente à la conception continue et au fait que le modèle Kevoree définisse des entités dynamiquement adaptables qui échangent les modèles dans lesquels ils sont eux-même représentés, apparaît un nouveau problème que l'on nomme *hara-kiri*. En effet, par analogie avec le suicide des samurais japonais [10] les groupes et les nœuds manipulent des modèles qui peuvent eux-mêmes les remettre en cause et donc causer

10. http ://fr.wikipedia.org/wiki/Seppuku

FIGURE 6.11 – Fragmentation des groupes



leur arrêt. Par exemple, si un groupe reçoit un modèle dans lequel celui-ci a subi une mise à jour du binaire, après transfert au M@RC ce dernier va irrémédiablement causer l'arrêt du groupe pour la mise à jour de son binaire. Un exemple similaire peut illustrer la mise à jour d'un nœud instance *via* un *hara-kiri*.

Le problème se pose majoritairement dans le cas où les algorithmes des groupes nécessitent des appels synchrones au M@RC, par exemple en s'inscrivant après la mise à jour (post des *ModelListener*). C'est ce cas de figure qui intervient lorsqu'un groupe de type consensus doit être supprimé dynamiquement d'un nœud (par exemple pour être remplacé par un *gossip*). Dans ce cas de figure, s'il est possible que les groupes définissent un consensus sur le nouveau modèle (phase 1) il est impossible de le faire après la mise à jour (phase 2).

Pour faire face à cela le groupe doit détecter ce cas de figure, supprimer ses *ModelListener* et transformer ses appels au M@RC synchrones en asynchrones. Pour résoudre cette détection, les groupes et nœuds instances peuvent exploiter la réflexion qu'apporte le nouveau modèle sur le futur état. Ainsi la solution apportée par le modèle Kevoree est un vérificateur de *hara-kiri* offert aux instances pour étendre leurs algorithmes et tenir compte de cette spécificité d'usage. Celui-ci est formalisé comme suit :

Déf 15. $HaraKiri : (oldInstance : Instance, oldModel : KevModel, newModel : KevModel) \rightarrow Bool \rightleftharpoons \nexists el.(el \equiv oldInstance) \vee \exists el.(el = oldInstance \wedge \nexists el \equiv oldInstance)$

Une instance subit donc un *hara-kiri* si elle n'existe plus dans le nouveau modèle ou si son équivalente (identification équivalente) a subi une mise à jour de binaire.

6.4.4 Consensus du DDAS sur un modèle : groupe Paxos

Comme introduit dans la sous-section 2.3.8 les algorithmes de consensus et particulièrement les protocoles *Paxos* visent à garantir la cohérence d'une donnée répliquée sur plusieurs nœuds distribués. L'exigence d'un consensus majoritaire a deux buts principaux : garantir la séquence de mise à jour des nœuds et surtout garantir la pérennité de la donnée en assurant qu'un certain nombre de nœuds font partie du quorum d'acceptation.

Les groupes du modèle Kevoree rentrent typiquement dans la terminologie *Paxos*, la donnée à ordonner et à garantir est alors le modèle structurel de l'application. Ainsi les algorithmes de consensus sont particulièrement adaptés lorsque l'on veut assurer un état et donc un modèle cohérent pour un ensemble de plates-formes. Par transposition, l'application d'un algorithme *Paxos* dans le cas du M@R se ramène à l'obtention d'un consensus sur la prochaine valeur que doit prendre le modèle avant son application locale.

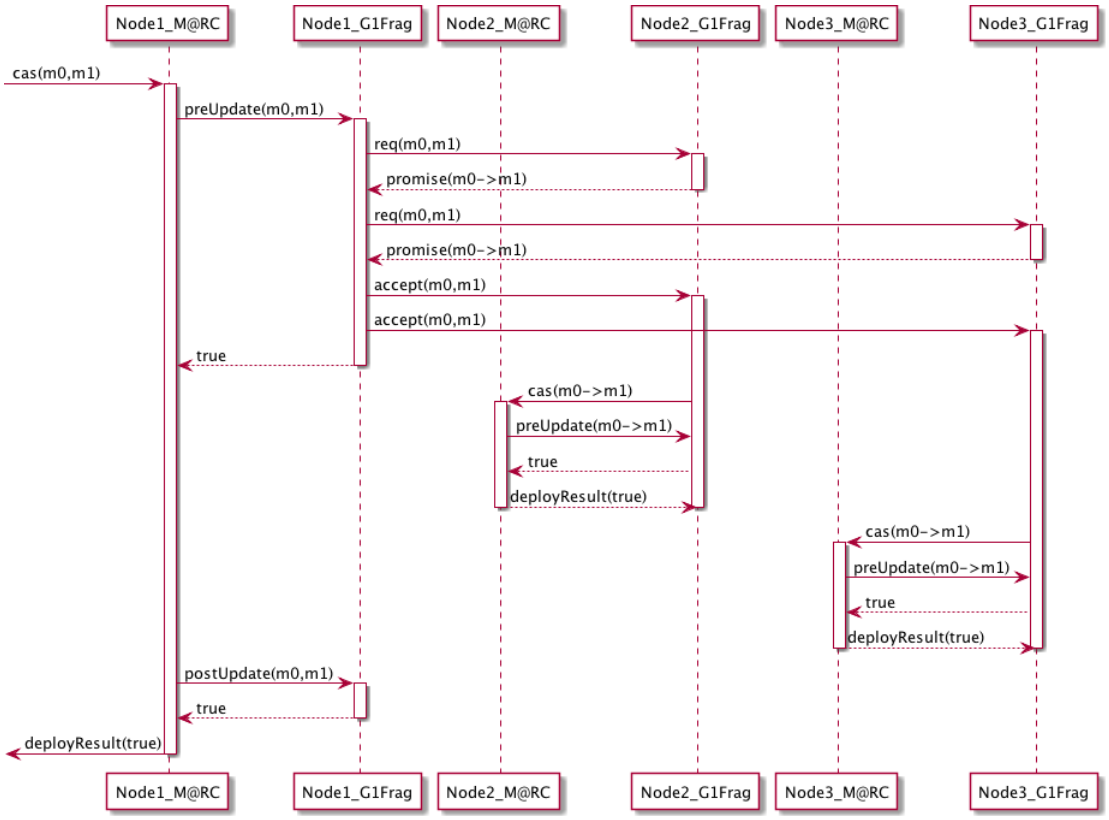
On définit donc un *GroupType* Kevoree qui exploite un algorithme basique de *Paxos*, ce dernier doit permettre d'obtenir une garantie de cohérence d'un ensemble de nœuds de sorte que le DDAS global garantisse la sauvegarde de son état même en perdant un grand nombre de nœuds. On fait donc le parallèle entre la terminologie *Paxos* et Kevoree .

- Le rôle *Client* de *Paxos* est pris ici par le M@RC et plus précisément *via* son appel à la méthode *preUpdate* qui doit ou non permettre une mise à jour locale. Le client doit proposer une nouvelle valeur qui prend ici la forme d'un modèle.
- Les rôles *Processor* de *Paxos* sont pris par chacun des fragments d'un groupe déployé. Chaque fragment étant lié a un M@RC, on considère qu'ils ont chacun une copie du modèle et donc de la valeur du consensus.
- Les *Quorum* sont calculables à l'aide des éléments présents dans la relation *subNodes* du groupe déployé. Ainsi suivant le taux de majorité voulu les *Quorum* correspondants peuvent extraire des sous-ensembles de cette relation.
- Le *Proposer* est ici le fragment qui a détecté la mise à jour *via* une interception à la méthode *preUpdate* du M@RC.
- Les rôles *Learner* et *Leader* sont pris par un ou plusieurs des fragments suivant les variations de *Paxos* visées.

Dans sa version la plus basique, le groupe *Paxos* mixe les rôles de *Proposer* et de *Leader/Learner*. Le principe est le suivant : un nœud voulant faire une mise à jour locale doit obtenir l'aval d'une majorité de nœuds avant de déclencher ses adaptations locales. Cette majorité obtenue pour une migration d'un modèle V0 vers une version V1, il déclenche sa mise à jour et envoie un message d'acceptation aux autres nœuds pour qu'ils fassent de même. Les *Quorum* d'accepteurs sont cependant calculés à partir de la version courante du modèle et non du modèle à propager. En effet, pour les mêmes raisons que le hara-kiri il sera impossible d'obtenir un consensus avant le premier déploiement des nouveaux fragments du groupe. La valeur d'initialisation peut soit venir d'un autre groupe soit suivre une acceptation majoritaire comme préconisé dans la référence [Lam01]. La figure 6.12 illustre le fonctionnement nominal de cette migration de modèle V0 vers V1 suivant la topologie du groupe G1 de la figure 6.11. Les acteurs du diagramme de séquence sont nommés à l'aide du nom du nœud qui les hébergent plus _M@RC pour le M@RC ou _G1Frag pour désigner le fragment local du groupe G1. Les appels de méthodes du *CompareAndSwap* du M@RC sont notés *cas*.

Une fois la majorité de message *accept* collectée, N1 peut déclencher sa mise à jour en retournant *true* à son M@RC, après avoir libéré les autres nœuds du groupe, qui déclenchent alors un *compare and swap* sur leurs M@RC respectifs. Cette version peut être étendue avec l'élection d'un leader ou de plusieurs *learners* pour résister à des

FIGURE 6.12 – Diagramme de séquence : Kevoree basic Paxos



attaques byzantines [CL98].

Il est à noter que les modèles structurels sont une donnée relativement volumineuse ; ici plus qu'ailleurs la consommation réseau d'un tel mécanisme est importante et nécessite des connections *multicast* pour être optimale. L'objet de cette contribution étant centré sur l'usage du M@RC, l'ensemble des dérivations possibles de *Paxos* qui n'influe que sur les échanges entre groupes fragments n'est pas détaillé ici. A l'inverse le reste de cette section s'attarde sur les mécanismes possibles entre différentes instances de groupes consensus au sein d'une même plate-forme.

6.4.5 Consensus de migration ou consensus de mise à jour

Les *ModelListeners* et donc plus généralement les groupes Kevoree peuvent exploiter deux phases d'interception du M@RC : (pre et post) *update*. Le *basic Paxos* défini précédemment exploite uniquement l'interception pré-déploiement, le consensus porte donc sur l'acceptation d'une migration d'un modèle à un autre et non son déploiement global (m0->m1).

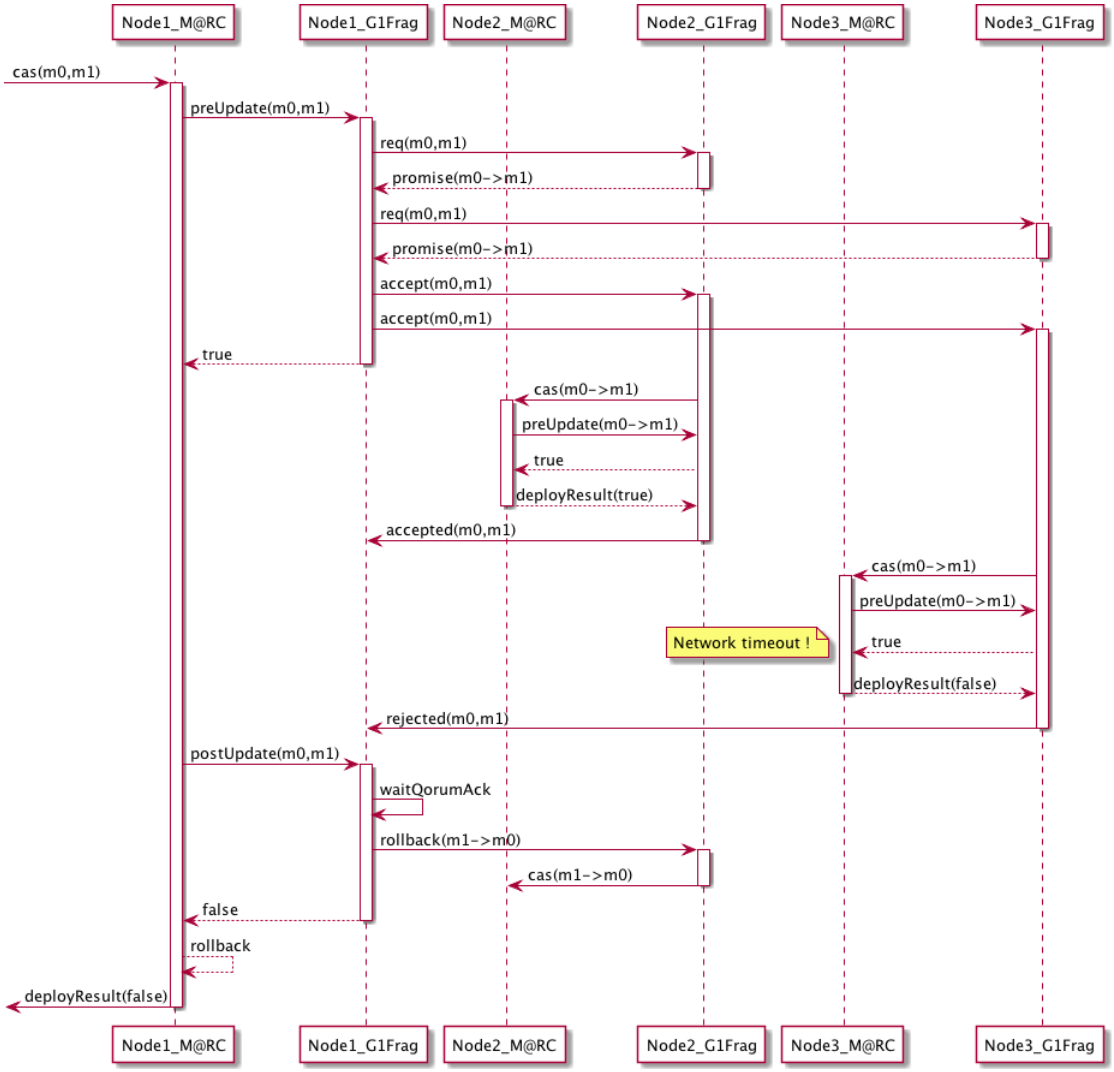
Ce mécanisme peut être étendu en exploitant non plus uniquement l'interception

pre mais également l'étape *post* pour introduire une étape de collecte des résultats de déploiement sur le quorum d'acceptation de la première phase du *Paxos*. Cette deuxième phase n'est pas à proprement parler un *multi-Paxos* mais le but est bien de réexploiter le leader de la phase précédente afin de collecter les résultats après les déploiements distants.

Dans le cas où le taux d'acceptation est insuffisant, un *rollback* est demandé localement et à distance pour revenir à la version précédente du modèle. Le taux d'acceptation peut correspondre soit à la totalité du quorum soit à une fraction, suivant le cas d'usage.

La figure 6.13 illustre par un diagramme de séquence la séquence du protocole avec un fragment de groupe défectueux.

FIGURE 6.13 – Diagramme de séquence : Kevoree two step Paxos



Dans cette séquence l'erreur d'un seul des éléments du quorum ayant accepté la première phase déclenche un *rollback* distribué. L'erreur dans ce cas intervient lors du *compare and swap* du M@RC du nœud N3 pour une erreur de *timeout* lors du téléchargement de binaire.

6.4.6 Groupe exclusif ou lock-free

Les erreurs de déploiement peuvent intervenir soit pour des raisons contextuelles de déploiement dues au code à exécuter mais aussi à cause d'accès concurrents sur le M@RC. En effet ceci illustre un problème qui intervient lorsque plusieurs instances de groupes sont déployées pour un nœud. Dans ce cas de figure plusieurs fragments interagissent avec le M@RC et peuvent alors faire échouer le *compare and swap* si entre le consensus d'acceptation de migration et le déploiement effectif un autre groupe réussit à déployer un autre modèle. Prenons un exemple sur l'architecture de la figure 6.11. Lors d'une exécution d'une migration le nœud N3 et son groupe G1 accepte un consensus de migration de V0 vers V1, entre temps le groupe G2 reçoit une mise à jour de V0 vers V3 qui s'exécute avant le CAS de G1. Dans ce cas le CAS(m0->m1) va échouer puisque la valeur initiale a été changée.

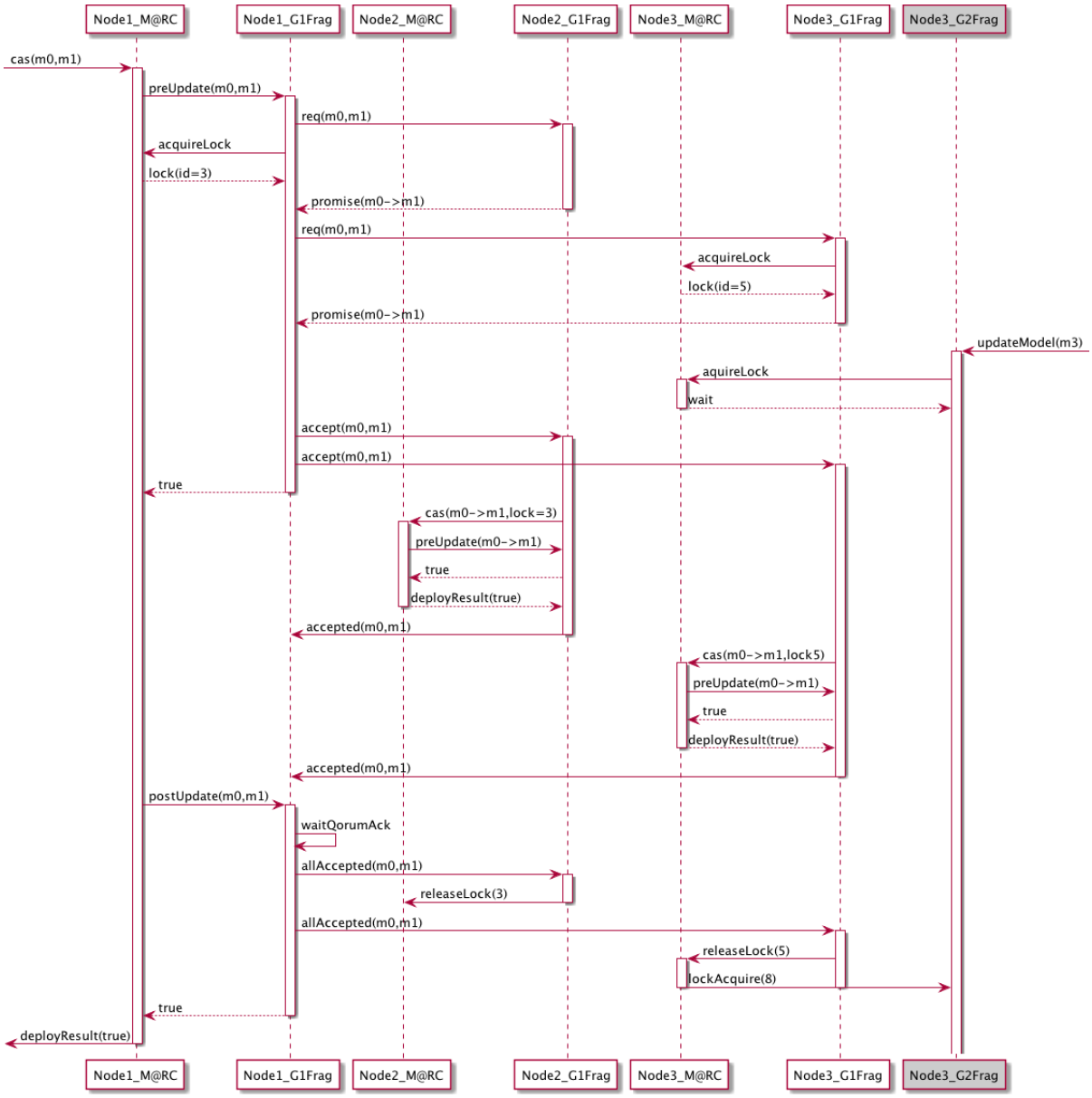
Comme pour le traitement des accès concurrents aux mémoires partagées les groupes Kevoree peuvent exploiter deux approches pour résoudre ce problème :

- Une version optimiste qui ne bloque pas le M@RC et fait confiance aux opérations CAS pour détecter les erreurs. Cette famille correspond à la famille des algorithmes *lock-free* , qui offrent de bonnes performances et surtout ne bloquent jamais les exécutions des autres groupes qui voudraient interagir avec le *Core*. Cette version optimiste produit cependant plus d'erreurs de déploiement et peut forcer des mécanismes d'adaptation à rejouer leurs algorithmes pour cause de collision.
- Une version exclusive qui pré-réserve le *Core* pendant la phase de *pre* et interdit donc tout accès pendant cette réservation. Cette méthode à verrou permet d'assurer le déploiement une fois le consensus accepté par le quorum mais bloque les autres groupes ou raisonneurs locaux tant que le verrou n'est pas levé. Ce type d'accès exclusif permet de limiter le besoin de rejeu des algorithmes producteurs de modèle.

La version précédente de *Paxos* peut donc encore être étendue avec des pré-réservations. En d'autres termes avant d'envoyer leurs messages *promises* les *acceptors* vont poser un verrou sur le *Core*, qui leur permettra de conserver un accès exclusif et d'être les seuls à pouvoir passer la barrière du CAS ou *update* local sans blocage. Les autres processus qui veulent accéder à la demande de verrou, au *compare and swap* ou à une méthode *update* de modèle vont être mis en attente. La figure 6.14 définit le diagramme de séquence de ce nouveau groupe en illustrant un cas de collision entre le groupe G1 et G2, toujours sur la topologie de la figure 6.11. Cette collision peut s'illustrer par une demande de verrou de G2 ou même directement une demande de *compare and swap* de G2.

La pose de verrou garantit donc que le groupe G1 poussera son déploiement jusqu'au bout mais a un impact sur le groupe G2.

FIGURE 6.14 – Diagramme de séquence : Kevoree Paxos exclusif



6.4.7 Groupe pour les réseaux P2P : association de Gossip et Vector-Clock

6.4.7.1 Objectifs et spécificité d'une dissémination P2P

Afin d'assurer un ordre total, les consensus imposent de manière implicite certaines contraintes pour être exploités de manière optimale. D'une part, les communications

réseaux doivent être directes, entre le fragment origine de la propagation et les *acceptors*. Idéalement, le type de réseau doit supporter des communications *multicast* pour réduire les envois simultanés des nouveaux modèles et ainsi réduire la charge du réseau. D'autre part l'essence même du consensus est d'assurer la cohésion d'un groupe de nœuds lorsque sa majorité est présente, ce type d'approche est donc nécessairement dédié aux clusters dont les connexions sont raisonnablement persistantes. Si la fiabilité de connexion est trop faible les mises à jour sont simplement bloquées. Enfin, le consensus nécessite une réflexivité exhaustive sur la topologie réseau et cette topologie doit refléter au plus près le système réel. En effet en cas de divergence, par exemple si des nœuds sont marqués comme participant au consensus sans être encore synchronisés, le consensus va connaître un fort taux d'échec.

Si on reprend le cas d'usage sapeur-pompier motivant ces travaux, on s'aperçoit à l'inverse d'un consensus qu'un réseau tactique de terrain doit fonctionner quelque soient les aléas et le taux de perte des nœuds. Dans ce cas d'usage et dans beaucoup d'autres l'hypothèse de connexion persistante n'est plus envisageable, de même que les blocages en cas de plan d'urgence et de défaillances de nœud trop nombreuses.

Pour ce cas d'usage il est important de disposer d'une propagation plus épidémique et plus opportuniste telle que celle proposée par les protocoles *gossip*. Cette pandémie peut venir de n'importe quels nœuds puisque chacun peut avoir la liberté d'héberger des raisonneurs capables de modifier potentiellement tous les DDAS. Cette sous-section détaille l'usage de protocole *gossip* en tant que groupe Kevoree et donc dédié à la propagation épidémique de modèle. Ainsi dans cette approche chaque fragment d'un groupe *gossip* est dédié à propager de façon épidémique ses modifications aux autres fragments. Pour cela il va échanger ce modèle avec un de ses voisins, qui fera alors de même de proche en proche jusqu'à la convergence du *cluster*.

6.4.7.2 Une architecture moyenne calculée comme une agrégation épidémique *gossip*

Dans ce mode de propagation aucun ordre n'est assuré pour les mises à jour, et aucun verrou n'est posé. Un problème intervient cependant, lorsque deux nœuds calculent deux modèles différents à chaque extrémité du réseau P2P : ces modèles sont propagés de façon concurrente. De même lorsque qu'un réseau P2P possède des sections isolées pour cause de perte de communication, chaque sous-réseau va évoluer et donc faire émerger son propre modèle. Lorsque ces modèles concurrents entrent en collision il faut alors prendre une décision pour la réconciliation de ces informations.

Dans l'article [JMB05], Jelasity *et al.* proposent une agrégation épidémique fondée sur un protocole *gossip*. Le principe est le suivant. Chaque nœud a besoin d'une valeur agrégée du *cluster* pour prendre des décisions mais l'appel à tous les nœuds est trop coûteux. A l'inverse, chaque nœud fait une moyenne avec ses voisins à l'aide d'échange *gossip* pour maintenir une valeur la plus proche possible de celle contenue en pratique dans le *cluster*. Dans l'article [JB06a], les auteurs ont appliqué le même principe dans le projet *T-man* pour calculer une topologie de façon empirique.

Le besoin de valeur pondérée est transposable aux modèles structurels. En effet,

lorsque deux modèles sont concurrents on peut réaliser une opération de fusion afin de produire un modèle agrégeant les données des deux parties. À la manière de l'agrégation épidémique proposée par Jelasity, la contribution de ce groupe *gossip* est d'assurer un modèle moyen du DDAS. Cependant à la différence d'une valeur chiffrée, les modèles doivent détailler d'avantage d'information sur leur provenance, afin de détecter les conflits et pouvoir faire la réconciliation à la manière d'un gestionnaire de version de code source tel que *git* [11].

6.4.7.3 Principe de combinaison des horloges vectorielles ainsi qu'une propagation *gossip*

Le principe général de la solution proposée dans ce *group gossip* Kevoree est de combiner une propagation à la *gossip* avec des méta-informations pour être capable de détecter les branches divergentes et les réconcilier. Ainsi à la manière d'un gestionnaire de version, les modèles échangés doivent être équipés d'horloges logiques pour détecter ces branches. Cette solution a été publiée à la conférence DAIS'2012 [FDP+12b].

Les horloges logiques sont une solution très utilisée pour ordonner des échanges asynchrones non ordonnés. Lamport *et al.* [Lam78] proposent dans un article de 1978 d'ajouter un temps logique à chaque envoi de message, ce temps logique doit être partagé par l'ensemble des nœuds *via* une phase d'initialisation. Peu après en 1988, Fidge *et al.* [Fid88] et Mattern [Mat89] proposent conjointement d'exploiter un vecteur de temps logique associé à chaque évènement envoyé. Plus décentralisée cette solution n'impose pas d'horloge logique commune puisque l'horloge vectorielle embarquée dans chaque message contient les temps logiques des horloges de chaque nœud traversé par le message. L'horloge vectorielle sert alors à la traçabilité du message et surtout à sa synchronisation sur chaque nœud. Cette solution a été très utilisée par la suite pour gérer des ordres partiels d'évènements [BR02] ou même des synchronisations de base de données *clé/valeur* distribuées, tel que le projet Voldemort [12] de LinkedIn.

6.4.7.4 Protocole *gossip* pour dissémination de *Model@Runtime*

Le protocole *gossip* de ce groupe échange donc des *VectorClocks* ainsi que des modèles accompagnés de *VectorClock* . Le listing 6.8 définit en Scala ces trois structures, l'horloge logique est représentée par un *Long* , un *VectorClock* est donc essentiellement une liste de *ClockEntry* associant un nœud à une horloge. Le *nodeID* identifie ici le fragment du groupe courant déployé sur le nœud dont le nom correspond à ce *nodeID* .

Listing 6.8– Définition des VectorClocks

```
case class ClockEntry(nodeID:String, version:Long)
case class VectorClock(entries:List[ClockEntry])
case class VectorClockModel(model:KevModel, vectorClock:VectorClock)
```

11. http ://git-scm.com
12. `http://project-voldemort.com`

Les modèles d'architecture représentant un volume de données important, la propagation *gossip* naïve de cette donnée coûterait beaucoup trop cher en terme d'occupation réseau. La solution proposée vise donc à découper le protocole en deux phases, une phase de diffusion des *VectorClocks* de chaque nœud correspond à un modèle, et un échange entre deux lorsque l'échange de modèle est nécessaire. Voici les grandes lignes de cette solution.

- Diffusion des *VectorClocks* en associant une approche pro-active et réactive. Les nœuds se synchronisent de façon périodique avec leur voisin mais exploitent également un mécanisme pour notifier leurs voisins en cas de modifications. L'association des deux mécanismes permet à la fois de supporter un fort taux d'échec de communication tout en garantissant un temps de propagation court pour les nœuds disponibles sans erreurs.
- Communication inversée de la donnée à transférer (modèles) suivant le principe d'Hollywood [Fow04] et donc avec une inversion de contrôle. Là encore ceci est fait pour minimiser les échanges réseaux, les modèles ne sont jamais envoyés spontanément sur le réseau mais toujours demandés activement par un tiers.
- Le *gossip* est alimenté par ce qu'il dissémine (sa charge utile), c'est-à-dire le modèle. En effet les services essentiels que sont le *PeerSampling* et le *PeerSelector* se font en exploitant la couche de réflexion du modèle.
- Le service de *PeerSelector* exploite un historique d'état des communications en supplément de la topologie réflexive du modèle afin de réduire les tentatives de communication vers les nœuds en faute et ainsi réduire leur impact sur le fonctionnement du DDAS.

Cet algorithme a fait l'objet d'une validation expérimentale, détaillée dans la partie III validation. La partie 1 définit le vocabulaire des différentes données échangées durant un tel protocole.

Algorithm Part 1 DEFINITIONS

Message ASK_VECTORCLOCK, ASK_MODEL, NOTIFICATION
Type VectorClockEntry := <id : String, version ∈ ℕ>
Type Node // represents a node on the system
Type Model // represents a configuration of the system
Set Group := {node : Node}
Set IDS(g : Group) := {id : String | ∃ node : Node, node ∈ g & node.name = id}
Set Neighbors(originator : Node, g :Group) := {node : Node | node ∈ g & originator ∈ g}
Set VectorClock(originator : Node, g : Group) := {entry : VectorClockEntry | entry.id == originator.name}
1: ⋃ {entry1 : VectorClockEntry |∃ node : Node, node != originator & entry1.id ∈ IDS(g) & node ∈ g}
Set VectorClocks(originator : Node, g : Group) := {vectorClock : VectorClock(originator, g)}

Un protocole *gossip* est dédié à la dissémination d'un état. Chaque fragment d'un groupe *gossip* est donc un participant à ce protocole et stocke donc un état qui lui est propre. Cet état est majoritairement contenu dans le M@RC avec l'accès au modèle courant mais nécessite également des informations supplémentaires (voir partie algorithme 2). Chaque fragment stocke donc son *VectorClock* courant, un score associé à chacun de ses voisins pour les besoins du service de *PeerSelection* .

Algorithme principal (voir partie algorithme 3) Le protocole proposé est asynchrone et sans état, cependant on peut tout de même parler de cycle pour plus de compréhension. Un cycle *gossip* correspond à l'envoi d'une requête d'état vers un voisin, une comparaison de cet état avec le local et une synchronisation si nécessaire avec le nœud voisin.

Comme tous les *ModelListener* les fragments de groupe *gossip* sont notifiés à chaque changement local du modèle courant, une notification est donc envoyée à tous les voisins directs pour déclencher un cycle *gossip* anticipé chez ces voisins. En retour en début de cycle de protocole les autres fragments vont envoyer un message de requête d'état (structure *ASK_ VECTORCLOCK*) au fragment origine de la modification. Comme les connexions sont volatiles et non sûres, des notifications peuvent être perdues et non reçues par des membres du groupe. Pour faire face à ces pertes, chaque fragment déclenche également un cycle du protocole périodiquement en choisissant alors un voisin cible de la synchronisation *via* la méthode *SelectPeer*. Les modèles étant une donnée d'un volume conséquent, le protocole fait d'abord une demande du *VectorClock* seul puis une demande du *VectorClock* , accompagné du modèle si la synchronisation est nécessaire.

Pour déterminer si la synchronisation est nécessaire, le protocole repose sur une comparaison des *VectorClocks* détaillée en annexe 1. De façon très simplifiée cette comparaison évalue deux à deux les entrées des *VectorClocks* . Si un nouveau nœud est présent dans le *VectorClock* distant, ou si une version d'une de ses entrées est supérieure au local il est déclaré comme étant postérieur. A l'inverse si un nœud est manquant dans le distant ou si une version est inférieure il est déclaré comme antérieur. Si toutes les versions ne sont pas strictement égales ou supérieures le *VectorClock* distant est déclaré comme concurrent.

À la réception d'un *VectorClock* , un fragment effectue donc la comparaison avec sa copie locale. Si le *VectorClock* reçu est antérieur il est simplement ignoré et le cycle est fini. S'il est postérieur ou concurrent, une demande de *ASK_ MODEL* est envoyée. Lorsque qu'un modèle accompagné d'un *VectorClock* est reçu par un fragment il refait une comparaison, et si le modèle est toujours postérieur il fait la mise à jour avec ce nouveau modèle. Si les deux modèles sont toujours concurrents cela signifie que le *VectorClock* local et le distant ont des modifications concurrentes et proviennent donc d'une divergence. Afin de garantir la pérennité des modifications le fragment local fait une fusion (*merge*) des *VectorClocks* ainsi que du modèle avant de faire la mise à jour locale. La fusion des modèles peut être soumise à des conflits qui peuvent être résolus à l'aide de règles de priorité suivant les cas métiers (par exemple en donnant la priorité

Algorithm Part 2 STATE

```
1: localFragment : Group // pointer to local group instance fragment
2: currentModel : KevModel // local version of system configuration
3: localNode : Node // representation of local node instance
4: currentVectorClock ∈ VectorClocks(localNode, g)
5: scores := {<node : Node, score>, node ∈ Neighbors(localNode, g) && score ∈ ℕ}
6: nbFailure := {<node : Node, nbFail>, node ∈ Neighbors(localNode, g) && nbFail ∈ ℕ}
```

à la donnée passée par des nœuds maîtres).

L'algorithme 3 décrit le pseudo-code de ce protocole.

Algorithm Part 3 ALGORITHM

```
On init() :
 1: vectorClock ← (localNode.name, 1)
 2: scores ← {Neighbors(localNode, g) × {0}}
On change (currentModel) :
 3: incrementLocalVectorClock()
 4: ∀ n, n ∈ Neighbors(localNode, g) → send (n, NOTIFICATION)
Periodically do() :
 5: node ← selectPeerUsingScore()
 6: send (node,ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode, g), NOTIFICATION) :
 7: send (neighbor, ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode,g), remoteVectorClock ∈ VectorClocks(neightbor, g)) :
 8: result ← compareWithLocalVectorClock (remoteVectorClock)
 9: if result == BEFORE || result == CONCURRENTLY then
10:     send (neighbor, ASK_MODEL)
11: end if
On receive (neightbor ∈ Neighbors(localNode,g),vectorClock ∈ Vectorclocks(neighbor, g), model)) :
12: result ← compareWithLocalVectorClock (targetVectorClock)
13: if result == BEFORE then
14:     updateModel(model)
15:     mergeWithLocalVectorClock(vectorClock)
16: else if result == CONCURRENTLY then
17:     resolveConcurrently(vectorClock, model)
18: end if
On receive (neighbor ∈ Neighbors(localNode,g), request) :
19: if request == ASK_VECTORCLOCK then
20:     send (neighbor, currentVectorClock)
21: end if
22: if request == ASK_MODEL then
23:     send (neighbor, <currentVectorClock,currentmodel>)
24: end if
```

Fonction *SelectPeer* (voir Algorithme 4) La fonction *SelectPeer* est appelée périodiquement pour choisir un nœud pour lancer un cycle de *gossip* et se synchroniser avec lui. Ce service est essentiel pour assurer la qualité de la distribution uniforme des échanges, qui idéalement doivent s'organiser suivant une loi aléatoire [JVG+07]. Différentes implantations de ce service sont disponibles dans la littérature avec différentes propriétés de convergence. Celle proposée ici est donc bien évidement interchangeable avec une approche plus classique de *peer sampling* suivant une loi aléatoire.

La fonction présentée ici cherche une répartition uniforme dans le temps des nœuds joignables tout en réduisant l'impact des nœuds en faute en limitant leur taux de synchronisation. Les liens avec les nœuds fils sont parcourus *via* les *NodeLink* orientés et sauvés dans le modèle. Les *NodeLink* orientés forment donc un graphe orienté qui répond aux besoins de topologie pour les réseaux P2P non uniformes comme expliqué par Kermarrec *et al.* [KvS07].

En cas d'échec lors de la sélection d'un nœud, son score augmente de deux fois son taux d'échec précédent. Le score d'un nœud est remis au minimum des scores des nœuds lorsqu'une communication est établie après un échec. Le score d'un nœud augmente de

un lorsqu'une synchronisation est effectuée. Ce mécanisme laisse donc une chance aux nœuds non joignables de revenir dans le processus mais rend prioritaires les nœuds joignables donc la dernière synchronisation est plus ancienne que les autres. Le listing 4 détaille cette fonction.

Algorithm Part 4 SelectPeer

```
Function selectPeerUsingScore()
 1: minScore := ∞ ; potentialPeers := {}
 2: for node → Neighbor(localNode, g) do
 3:     if node ! = localNode && getScore(node) < minScore then
 4:         minScore := getScore(node)
 5:     end if
 6: end for
 7: for node → Neighbor(localNode, g) do
 8:     if node ! = localNode && getScore(node) == minScore then
 9:         potentialPeers := potentialPeers ⋃ {node}
10:     end if
11: end for
12: node := select randomly a node from potentialPeers
13: updateScore(node)
14: return node
Function getScore(node ∈ Neighbors(localNode, g))
15: return scores(node)
Function updateScore(node ∈ Neighbors(localNode, g))
16: oldScore := getScore(node)
17: scores := scores ⋃ {node, oldScore + 2 * (nbFailure + 1)} \ {node, oldScore}
```

Le *merge* de *VectorClock* garde les plus grandes valeurs de chaque entrée comme décrit par Fidge et Mattern [Fid88],[Mat89]. La fonction d'incrément du *VectorClock* local est détaillée dans l'algorithme 5.

Algorithm Part 5 FUNCTIONS

```
Function incrementVectorClock()
1: if changed == true then
2:     ∀ entry, entry ∈ currentVectorClock & entry.id == localNode.name ⇒ entry.v ← entry.v + 1
3:     changed ← false
4: end if
```

6.4.8 Propriétés attendues

L'algorithme proposé est issu d'une composition d'approche de diffusion *gossip* dont le mode de communication est inversé et dont les données échangées sont horodatées à l'aide de *VectorClock* afin de faire converger le système. De cet assemblage sont attendues plusieurs propriétés détaillées dans les sous-sections suivantes. L'objectif de ces propriétés est de borner l'algorithme précédemment défini en terme de complexité temporelle et quantitative en nombre de messages. La validation présentée en section 9 illustre les performances de cet algorithme de façon expérimentale.

6.4.8.1 Propriétés de convergence

Propriété 1. *Si deux nœuds produisent deux modèles d'architecture incompatibles alors au bout d'un temps fini un modèle compatible est produit et diffusé à l'ensemble des nœuds du système.*

La définition précédente de cohérence des modèles en chaque nœud est assurée par deux mécanismes :

- en premier lieu, par la construction d'un ordre partiel distribué sur les modèles, qui sont étiquetés par une horloge vectorielle : un modèle m_2 dépendant causalement d'un autre plus ancien m_1 est compatible avec m_1 ;
- en second lieu, en cas d'absence d'ordre causal entre deux modèles, par la résolution de conflits assurée par un algorithme de calcul différentiel de modèles produisant un modèle compatible.

6.4.8.2 Complexité temporelle

La propagation d'un nouveau modèle depuis un nœud n vers tous les autres est proportionnelle au diamètre D du graphe constitué par le groupe, et au nombre moyen de voisins V dans ce graphe. En effet l'application d'un nouveau modèle au nœud n provoque l'envoi de messages de notification aux voisins de n, qui en réponse lui demandent son horloge vectorielle, constatent qu'elle est supérieure à leur propre horloge, demandent le modèle de n, mettent à jour leur modèle puis procèdent comme n. Il y a donc 5 échanges de messages entre voisins, et l'ensemble des nœuds est donc mis à jour en $(D-1)$ étapes. En l'absence de notification l'envoi de demande de *VectorClock* se fait périodiquement après un délai T établi comme propriété du groupe et donc partagé par toutes les répliques de l'algorithme proposé. Le pire cas temporel de cet algorithme correspond à la configuration sans notification et est donc proportionnel à la valeur périodique et au diamètre du graphe de nœud, la synchronisation se fait alors en 4 étapes seulement. Soit T_M le délai moyen d'envoi d'un message, dans le pire cas des cas le temps total de mise à jour est inférieur ou égal à $(D-1) * V * T * (4 * T_M)$. Le meilleur cas correspond à une sélection immédiate du voisin, ce qui donne une borne inférieure de $(D-1) * T * (4 * T_M)$ sans emploi de notifications. Dans le cas où les notifications sont employées la borne inférieure devient $5 * T_M$.

Les résultats présentés en section 9.3.1 de cette thèse montrent une évaluation expérimentale de cette complexité sur un cas d'usage de topologie mobile. En pratique également la désactivation des notifications et donc le mode *pull* seul n'est pas une solution satisfaisante. Cependant à la manière du protocole T-Man [JB06a], il est envisageable d'envoyer des notifications à un nombre contrôlé de noeuds, garantissant la non-inondation du réseau , tout en garantissant la convergence. Cet envoi de notification de manière non aveugle n'est pas détaillé ici mais peut largement profiter de l'approche Models@Runtime et de son historique pour par exemple sélectionner les nœuds les plus régulièrement mis à jour.

6.4.8.3 Propriétés de résilience à l'intermittence des connexions

Propriété 2. *Toute coupure de communication entre deux nœuds voisins entraîne une forte probabilité de création d'un sous-réseau et donc d'une divergence de deux modèles évoluant de manière incompatible. Plus la résolution est faite rapidement plus elle est considérée comme simple.*

Les évènements de reconnexion sont exploités afin d'améliorer cette propriété.
- Dès sa reconnexion un nœud déclenche un cycle de *gossip* avec le voisin dont la dernière synchronisation est la plus ancienne.
- À l'inverse toute reconnexion d'un voisin précédemment en erreur entraîne une augmentation de sa priorité pour le prochain cycle. En revanche, si aucune reconnexion n'est observée le noeud est progressivement éliminé de la sélection.

Une validation expérimentale de cette propriété est proposée en section 9.3.3.

6.4.8.4 Complexité en nombre de messages

Propriété 3. *Les échanges de modèles sont considérés comme plus coûteux que les envois de notification et de* VectorClock .

Soit N le nombre de nœuds du graphe liés au groupe considéré, et soit V le nombre moyen de voisins. Chaque nœud va chercher une stabilisation avec ses voisins, ce qui va nécessiter à chaque étape une notification et un échange de *VectorClock* . À cela s'ajoute un envoi de modèle lorsque la réception d'une horloge vectorielle montre qu'une synchronisation est nécessaire. Une borne supérieure du nombre de message est la suivante :

$$(N-1)*V*NOTIFICATION+ASK\_VECTOR\_CLOCK+VECTOR\_CLOCK$$
$$+(N-1)*(ASK\_MODEL+MODEL)$$

Dans le cas où le système de notification est employé, V prend la valeur 1, ce qui correspond également au meilleur cas sans notification. Que ce soit avec ou sans notification, l'envoi du modèle vers tous les noeuds est inévitable. L'usage d'un envoi en deux passes (horloges puis modèles) permet ainsi de réduire l'envoi de modèle sans raison, l'impact de ce choix sera évalué de façon expérimentale dans la section 9.3.1.

6.4.9 *Slicing* de modèle à l'image du *peer sampling*

Les propriétés décentralisées du *gossip* ont permis et amené son utilisation sur des réseaux P2P fédérant un très grand nombre de nœuds. La construction exhaustive de la topologie est déclarée comme non envisageable dans la littérature. Le *slicing* et le *peer sampling* sont alors la solution proposée pour extraire localement la liste utile et nécessaire des nœuds voisins pour les échanges *gossip*. La topologie est alors répartie en *slice* (sous-partie) sur différents nœuds.

La taille de la topologie due au nombre de nœuds n'est pas le seul problème, d'ailleurs dans le cas d'étude sapeur-pompier, les groupes n'atteignent pas une telle masse critique. Comme le rappelle Jelasity *et al.* dans un article sur le protocole *T-Man* [JB06a] et

dans [JVG+07] le problème n'est pas tant la taille de stockage de la topologie mais plutôt sa maintenance dans un système fédérant des nœuds très sporadiques qui se met perpétuellement à jour. En effet la mobilité et les connexions/déconnexions rapides des nœuds modifient la topologie de façon très rapide, de sorte que la maintenance globale est très difficile voire illusoire. Ce constat rejoint d'ailleurs celui fait pour les *Ultra-Large-Scale Systems* [NFG+06]. Une réponse à ce problème est la construction incrémentale proposée par le protocole *T-man*. Celui-ci construit la topologie de proche en proche tout en gardant sa visibilité dans des *slices* de nœuds. Ce type d'approche permet à la fois de découper la topologie est sous-parties mais également d'apporter une propriété de *self healing* ' celle-ci en auto-réparant les liens non valides.

De manière fonctionnelle, le *slicing* et le *peer sampling* visent à fournir un sous-ensemble *minimaliste et nécessaire* pour le fonctionnement d'une fonctionnalité. Cette fonctionnalité peut être par exemple un service de *SelectPeer* en assurant une répartition uniforme et aléatoire du nombre de synchronisation d'un nœud avec ses voisins. On peut alors faire le parallèle avec le modèle Kevoree qui dans notre approche sert de base topologique pour le *gossip*. Le *slicing* de modèle consiste également à le découper en sous-parties *minimalistes et nécessaires*. La portée des *slices* dépend de la fonctionnalité à garantir. Pour l'usage en tant que topologie du modèle Kevoree le besoin en connaissance topologique suit les mêmes règles que le *peer sampling*. La topologie Kevoree étant modélisée comme un graphe orienté tout comme les topologies exploitées en *gossip* [KvS07], les deux approches partagent les mêmes opérateurs de découpage.

Le concept *slicing* est lui même complètement cohérent avec le principe même du M@R. En effet le *slicing* est fait pour gérer la divergence de la topologie, tout comme le M@R qui lui cherche à gérer la divergence du modèle réflexif du DDAS. La capacité du *gossip* à fonctionner malgré cette divergence en fait un choix idéal pour la propagation du M@R.

La portée du *slicing* pour la partie modèle à composant de Kevoree est cependant plus délicate. La visibilité d'un *slice* dépend du besoin des nœuds en réflexion vis-à-vis des autres nœuds du cluster. Par exemple si des nœuds A et B ne sont pas adaptables par des nœuds B et C alors il n'est pas nécessaire de leurs transmettre la partie descriptive des composants de A et B. Le groupe *gossip* injecté par exemple entre ces nœuds peut alors couper le modèle (topologie et modèle de composant) pour n'envoyer que la partie minimale.

Cette technique est à opposer avec la solution visant à organiser les nœuds sous forme de hiérarchie dans le modèle *via* l'utilisation de plusieurs groupes. Sémantiquement les nœuds ne font pas alors partie d'un même groupe de synchronisation et exploitent alors des nœuds passerelles pour faire transiter les informations. La solution multi-groupes est donc plus adaptée pour un usage de Kevoree sur les réseaux maillés P2P hybrides tandis que le *slicing* est plus adapté pour un usage de Kevoree sur un réseau P2P pur.

En conclusion la capacité de divergence est un outil efficace et commun aux protocoles et aux modèles d'architecture pour répondre à la forte volatilité des nœuds participants.

6.5 Discussion sur l'impact du modèle Kevoree sur la construction du DDAS

Cette section conclut brièvement la contribution avant sa validation. Cette section détaille entre autres les points périphériques qui vont donner lieu à des perspectives à la fin de cette thèse.

Impact de la programmation pour la conception continue sur les raisonneurs

La contribution présentée ici vise à faire **remonter la conception des DDAS au niveau architecture**. La constitution de tels systèmes se fait alors en modélisant les flux d'échanges des modèles d'architectures Kevoree , mais également leurs étapes de transformation telles que le *slicing* de modèle.

A mi-chemin entre la *programmation par patrons* de Schmidt *et al.* et *celle par flux* tel que les EIP de Hohpe *et al*, le modèle Kevoree propose donc de gérer **l'hétérogénéité des synchronisations** DDAS comme un *flux d'échanges de modèles d'architecture.*

La description des modèles d'architecture reprend alors les résultats de la modularité des composants et la souplesse des architectures asynchrones inspirées par les usages réseaux des agents/acteurs. Mais au-delà du modèle structurel, l'approche préconise une isolation et responsabilisation des nœuds de calcul vis-à-vis de leurs états et adaptations de cet état.

Ainsi à la manière dont la programmation par objets délègue aux objets une responsabilité de traitement derrière des appels de méthodes, aux acteurs une responsabilité d'état et de processus derrière les envois de messages, Kevoree rend les nœuds responsables de leurs états vis-à-vis des modèles. Cette encapsulation toujours inspirée des patrons de découplages de la programmation par objets assure une délégation de traitement qui permet de faire face à **l'hétérogénéité des types de nœuds**. Cette encapsulation permet également de passer du modèle centralisé des superviseurs d'adaptation des grilles vers des DDAS ou **l'adaptation est totalement décentralisée** et peut alors se faire profitant de la robustesse des algorithmes épidémiques. Dans cette approche chaque nœud possède son indépendance et héberge son propre processus M@RC qui compose les raisonneurs locaux et les différents algorithmes de synchronisation.

L'approche de composition préconisée s'inspire des algorithmes *lock free* pour limiter les dépendances des noeuds du DDAS afin d'assurer une fiabilité accrue pour des réseaux non fiables. Ceci ne se fait pas sans un coût puisque cette approche introduit la possibilité d'échec lors de la recherche d'optimisation et d'auto-adaptation de ces plates-formes. Des mécanismes tels que le *compare and swap* permettent de gérer ces erreurs, mais il apparaît comme nécessaire de faire évoluer les processus de synthèse d'architecture vers un cycle à l'image de la plate-forme : continu. Ainsi pour faire face aux erreurs d'adaptation les mécanismes d'auto-adaptation qui devront diriger ces plates-formes devront être capables de rejouer leurs optimisations et ainsi introduire une notion optimisation incrémentale continue. Les différentes approches abordées à la suite de cette thèse pour répondre à ce besoin sont abordées dans la section validation.

Troisième partie

Validation

Less is more.
Ludwig Mies van der Rohe

La contribution de cette thèse est une abstraction proposée pour résoudre les problèmes de gestion des DDAS. La qualité d'une abstraction se mesure en fonction de sa capacité à résoudre un problème donné sans en introduire un nouveau par sa propre gestion. Comme le rappelle la citation de *Van der Rohe*, l'aspect synthétique d'une abstraction garantit sa compréhension et sa versatilité sur différents usages. Pour obtenir cette expressivité cette contribution exploite les outils de modélisation du design directement pour la gestion des plates-formes. Cette section cherche à valider que le surcoût introduit par cette abstraction rend réaliste son usage sur des cas d'usage aussi différents que ceux abordés dans le cas pompier servant de motivation. Ce chapitre traite également de la viabilité de l'usage d'outils du design directement dans les plates-formes. Après une analyse des axes de validation nécessaires en 7, une validation expérimentale est détaillée pour l'usage d'une telle abstraction dans un des cas limites sur des systèmes fortement contraints. Une validation expérimentale est par la suite proposée pour son usage sur des DDAS large échelle. Enfin la viabilité de la solution sur de multiples cas d'usage est abordée en détaillant l'état actuel du prototype mais surtout sa relation avec de nombreuses problématiques de recherche encore ouvertes sur les DDAS.

Chapitre 7

Axes d'évaluation

7.1 Une couche d'abstraction pour faire face à la complexité des DDAS

De manière très synthétique, la contribution de cette thèse propose de répondre à la complexité de gestion des DDAS en exploitant une couche abstraite pour en simplifier la manipulation. La montée en abstraction est une réponse communément exploitée pour le design de système large échelle ou critique au travers de solutions telles que le langage UML ou tout autre modélisation issue de l'IDM. Cette abstraction qui suit l'approche Model@Runtime exploite alors les méthodes à l'état de l'art de l'IDM directement dans les plates-formes. L'expressivité gagnée permet de reprendre tous les résultats de l'IDM [FDB+09],[MBJ+09b],[FHS+06],[CGFP09] en terme de gestion de variabilité de système ou encore de synthèse d'architecture. L'usage de techniques de modélisation a déjà été validé sur des systèmes raisonnablement puissants tels que ceux proposés dans la thèse de Brice Morin ou encore la dernière version de l'IDE Eclipse. Son usage sur des systèmes distribués hétérogènes implique l'inclusion de noeuds de plus faible puissance remettant alors en cause la viabilité du surcoût introduit.

7.2 Des outils du design au runtime, à quel coût ?

Les solutions sont multiples pour proposer des abstractions de développement, les API et autres approches par framework détaillées dans la partie contexte en sont des illustrations. Le choix d'une approche se fait alors en fonction de l'expressivité offerte et du surcoût introduit. L'approche proposée reprend les méthodes génératives de l'IDM basé sur un langage spécifique au DDAS. Par construction ce méta-modèle profite de l'expressivité de l'IDM pour l'ensemble d'outils de manipulation de modèle mais force également leurs usages ainsi que le portage du processus M@RC directement dans les plates-formes. Le modèle proposé est par définition extensible pour modéliser les différentes spécificités de chaque plate-forme. L'axe de validation critique n'est donc pas l'expressivité de la solution mais sa viabilité en terme de performance sur des systèmes contraints ou des systèmes large échelle.

7.3 Evaluation aux cas limites

Le cas pompier de motivation illustre parfaitement l'hétérogénéité et la distribution des systèmes DDAS visés. A défaut de pouvoir faire une étude exhaustive, cette validation de l'approche va donc s'intéresser aux cas limites identifiés dans le cas d'étude.

Le premier cas limite identifié est l'usage du M@R pour la gestion des types de noeuds les plus embarqués tels que ceux préconisés pour l'équipement de sécurité des pompiers. Si la délégation du M@RC est exploitable pour aider ces noeuds légers par d'autres plus puissants, les temps de reconfiguration ainsi que l'occupation mémoire reste un challenge. Ceci introduit donc la question suivante :

Question 1. *Les performances de l'adaptation dirigée par le M@RC Kevoree permettelle de respecter les contraintes de noeuds aussi hétérogènes qu'un cas d'usage Cyber Physical Systems (CPS) ?*

Une validation expérimentale dont les résultats ont été publiés à la conférence CBSE'2012 [FBP+12] traite de ces challenges en section 8. Les plates-formes exploitées sont suffisamment représentatives pour généraliser ces résultats aux différentes architectures embarquées qui participent elles-mêmes aux architectures de la classe des *Cyber Physical System.*

Le deuxième cas aux limites identifié est l'usage du M@R pour la gestion d'adaptation dans les réseaux pair-à-pair. Outre le problème de passage à l'échelle, la capacité du modèle d'architecture à servir de base réflexive pour des algorithmes tels que les Gossip est un challenge. Mais au delà, cette classe de problèmes met également à l'épreuve Kevoree pour résoudre les problèmes de divergence des DDAS. Cette divergence que l'on observe notamment dans les environnements à forte mobilité illustre parfaitement les problèmes rencontrés pour la construction des tablettes tactiles des systèmes tactiques de terrain des pompiers. De façon plus globale, ceci amène la question suivante :

Question 2. *Les groupes du modèle Kevoree permettent-ils de représenter et d'exécuter différentes stratégies de synchronisation, telles que la cohérence à terme ?*

Une validation expérimentale dont les résultats sont publiés à la conférence DAIS'2012 traitent de ces problèmes en section 9.

7.4 Evaluation de la généricité de l'approche

Kevoree est une approche outillée pour la réalisation des DDAS. L'expressivité d'une telle solution ne peut être évaluée qu'à partir de ses ramifications de projets liés afin de valider son usage dans différents cas d'étude. Chaque cas d'étude explore des problèmes différents inhérents à la construction dynamique des DDAS. Dans une dernière section, l'état de maturité du prototype sera évalué et une description des projets liés sera donnée pour évaluer cette versatilité de la solution.

Enfin les abstractions proposées dans ce modèle sont prévues pour faciliter le développement et la manipulation des DDAS par des utilisateurs humains. Kevoree a été

mis dans les mains de publics de développeurs très différents pour valider leur ressenti et leur compréhension d'une telle abstraction. Au delà des performances du modèle il est opportun d'évaluer son utilisabilité et sa versatilité ?

Question 3. *Le modèle Kevoree proposé joue-il son rôle d'abstraction des DDAS, et est-il maîtrisable pour les concepteurs de systèmes DDAS ? L'approche est-elle suffisamment générique et versatile pour adresser les différentes types de DDAS ?*

Le troisième axe de validation est organisé autour de ces deux questions. L'évaluation de l'utilisabilité de la solution nécessite un prototype concret dont l'implantation a fait l'objet d'une publication à la conférence MODELS'2012 [FNM+12]. Les langages et outils composants ce prototype ainsi que son évaluation face à des concepteurs types de DDAS seront détaillés dans la section . La versatilité du modèle sera argumenté autour d'une description en section des projets liés dans des domaines diverses, exploitant les capacités de Kevoree pour réalisation leurs adaptations dynamiques

Chapitre 8

Evaluation quantitative aux limites : gestion des environnements contraints

L'introduction d'une abstraction se traduit par un surcoût, de compréhension tout d'abord pour aborder les nouveaux outils mais, et c'est surtout sur ce point que se focalise cette approche un surcoût de fonctionnement de la plate-forme. Ce surcoût peut par exemple modifier le temps de réaction de la plate-forme dirigée par l'adaptation vis-à-vis d'une adaptation *ad hoc* ou encore être consommatrice de mémoire utilisée. Ce ralentissement peut être rédhibitoire pour un usage sur des nœuds dont la capacité est trop faible pour porter ce surcoût et ainsi être contradictoire avec la capacité du modèle Kevoree à gérer l'hétérogénéité. En effet si des approches *Model@Runtime* ont déjà été exploitées avec succès sur des nœuds de calcul puissants tels que ceux détaillés dans le projet européen Diva [MBNJ09a] ou même de façon plus commune dans des environnements de développements tels que Eclipse, il reste à valider cet usage sur des nœuds hétérogènes comprenant de fait des nœuds de plus faible puissance. L'objectif de cette section de validation est d'évaluer la viabilité de cet usage sur un cas limite d'usage du M@R avec l'adaptation sur des environnements fortement contraints.

Ce chapitre traite de cette évaluation en prenant l'exemple des nœuds capteurs de terrain du cas d'étude sapeur-pompier. Ces nœuds basse consommation et basse puissance sont le souvent utilisés en embarqué, par exemple dans le cas présent dans un équipement de sécurité incendie. Les résultats de cette section sont alors généralisables à des périphériques similaires et plus puissants.

Cette section détaille tout d'abord le rôle 8.1 et les attentes 8.2 d'un nœud embarqué Kevoree avant d'en extraire les métriques permettant une évaluation du surcoût introduit. L'évaluation expérimentale proposée effectue une série de mesures directement sur micro-contrôleur afin de connaître les limites d'utilisation de la solution.

8.1 Connecter les DDAS à leur contexte physique

8.1.1 Convergence de l'Internet des Objets et des Cyber Physical Systems

Les nœuds embarqués du cas sapeur-pompier sont une illustration d'une interconnexion plus générale des systèmes d'information avec leur environnement physique. Ainsi on observe depuis une vingtaine d'années un changement de l'Internet du contenu vers quelque chose de plus complexe, dynamique, et qui mixe l'Internet des personnes, services (Internet Of Services (IoS)) et des objets (IoT) [har].

Au delà d'une simple collecte de données physiques, ce rapprochement des processus de calculs informatiques et des processus physiques est la définition même d'un *Cyber Physical System*, exprimé entre autres par Edward A. Lee [Lee06a]. Le terme CPS désigne donc un système dans lequel il existe une boucle de rétro-action entre les processus physiques et logiciels, qui peuvent alors s'influencer mutuellement [Lee08]. Cette catégorie de systèmes dépasse la tâche initiale des systèmes informatiques qui était de traiter et de transformer des données. Les bénéfices de telles interactions plus riches avec le monde physique permettent la construction de nombreux systèmes tels que :

- la robotique distribuée pour des usages de télémaintenances ou télémédecines ;
- la cyber-défense, que ce soit au travers de *drones* ou autres véhicules autonomes, ou de systèmes tactiques militaires ;
- les immeubles intelligents, tels que les bâtiments ou les usines adaptant leurs périphériques en fonction de contexte interne, comme par exemple la présence ou non d'utilisateurs humains ;
- etc.

Les systèmes informatiques embarqués sont depuis longtemps confrontés à cette interconnexion directe avec le contexte physique. Pour cette raison, que ce soit pour des cas d'usage embarqués ou non, les abstractions permettant de construire les CPS se sont rapprochées de celles des logiciels embarqués. Cependant les cas d'usage des CPS apportent une dimension de distribution et donc de concurrence des traitements. Pour Lee *et al.* [Lee06a], ces nouvelles dimensions vont poser des problèmes vis-à-vis des méthodes de développement. En effet les logiciels embarqués sont le plus souvent développés et surtout testés en mode *boîte noire*. Ce développement clos permet par exemple de maîtriser de façon précise la notion de temps nécessaire pour la discrétisation du contexte physique, mais est antinomique avec un développement de systèmes CPS collaboratifs et distribués. Par exemple le développement d'un contrôle commande d'un avion se fait en évaluant la rapidité d'exécution d'un logiciel sur un processeur donné et en maîtrisant l'ensemble et le nombre des types d'entrées et sorties, en utilisant par exemple des langages synchrones [HCRP91], [PHP87].

Les CPS amènent donc un certain nombre de nouveaux défis, que ce soit pour la conception d'un système d'information ouvert, ou pour rendre celui-ci réactif et intelligent vis-à-vis de son contexte environnement, ou encore pour faire cohabiter des développements embarqués avec des développements distribués. Il est d'ores et déjà clairement identifié que le développement de tels systèmes nécessite des abstractions

et architectures dédiées [Lee08], [TGP08], [CAS08]. La cohabitation entre informatique embarquée et distribuée vient des infrastructures hybrides des CPS qui sont composées entre autres d'informatique conventionnelle mais également embarquée (*smartphone*, capteur, etc). Ces capteurs exploitent un ensemble de processeurs basse consommation et basse puissance tel que les micro-contrôleurs qui font traditionnellement l'objet de développement de logiciels embarqués dédiés, mais qui évoluent d'ores et déjà vers un contexte distribué avec des langages tels que *nesC* [GLVB+03]. C'est de l'intégration de cette dernière sous-partie des CPS que vient une grande partie de leur hétérogénéité, et fera donc l'objet de ce chapitre qui vise à valider l'usage de l'abstraction proposée Kevoree pour le développement de cette sous-partie des CPS. Les CPS nécessitent de nombreuses autres sous-parties telles les garanties temporelles d'exécution ou la réalisation de boucle de rétro-action qui s'apparentent à de l'intelligence distribuée, seul l'aspect traitant de l'introspection et l'application des adaptations sera analysé et évalué ici sur des environnements embarqués.

8.1.2 Des DDAS aux CPS

En de nombreux points, les DDAS partagent les problématiques des CPS, notamment les aspects de distribution et surtout d'adaptabilité. En effet, par nature sujets aux pannes car devant faire face à l'imprévisibilité du contexte physique, les CPS doivent être adaptables pour répondre dynamiquement aux besoins ou résister aux pannes. Ceci est d'autant plus marqué par leur caractère distribué qui les oblige à prendre en compte de nouveaux nœuds en cours de fonctionnement. Ce rapprochement est d'autant plus opportun, car des abstractions communes sont proposées pour répondre à leurs problèmes . Ainsi Lee *et al.* [Lee06a] préconisent entre autres pour le développement d'une abstraction dédiée aux CPS de :

- rendre prévisible et contrôlable les canaux de communication (appelés *pipeline* dans ce domaine) ;
- rendre contrôlable et explicite la notion de concurrence des traitements ;
- rendre le développement modulaire pour faire face à l'hétérogénéité du déploiement.

Ces besoins sont partagés avec les besoins identifiés des DDAS et ont pour cette raison été intégrés à l'abstraction Kevoree proposée. Le prolongement des DDAS sur des CPS et donc sur des nœuds de type micro-contrôleur est donc le sujet de l'évaluation de cette section, et notamment quant à la faisabilité de l'application d'une abstraction pour les DDAS sur les nœuds types des CPS. Ce prolongement signifie que les nœuds du CPS vont devoir prendre part à l'adaptation dynamique telle qu'on le préconise dans les DDAS. Si ces nœuds possèdent une capacité de calcul suffisante pour stocker leurs états sous forme de modèles, ils n'ont souvent pas celle suffisante pour réaliser le traitement complet du M@RC. Ils sont donc une parfaite illustration de la capacité de Kevoree à exploiter des nœuds délégués pour les nœuds de faible puissance. La volatilité des nœuds qui composent un CPS amène d'autres contraintes telles que la résilience des états et des configurations des nœuds. Cette section traite donc de l'évaluation de performance mais également de la garantie de ces propriétés standards du CPS avec une approche

de développement sur Kevoree .

8.2 Besoins spécifiques des systèmes adaptatifs contraints

Les nœuds embarqués des CPS imposent de nouvelles contraintes vis-à-vis des nœuds plus *conventionnelles*. En effet, outre leur capacité plus restreinte ces périphériques sont utilisés sur des usages et avec des technologies qui imposent de respecter certains délais pour les temps de réaction ou encore imposent de réduire les écritures sur la mémoire pour allonger leur durée de vie. Les axes suivants représentent alors les défis de l'application du M@R et plus généralement ceux de n'importe quelle adaptation dynamique sur ces nœuds embarqués avec peu de ressources :

1. **Temps d'arrêt** : Les micro-contrôleurs hébergent un micro-logiciel qui contrôle souvent directement les périphériques physiques. Redémarrer ou bloquer ces micro-contrôleurs peut avoir de graves conséquences s'il s'agit de contrôler des périphériques critiques, ou des conséquences indésirables il s'agit de contrôler des équipements de confort. L'adaptation doit donc limiter ce temps d'arrêt (*downtime*) autant que possible.
2. **Utilisation de la mémoire volatile** (RAM) : L'allocation dynamique de mémoire est la brique élémentaire de l'adaptation dynamique. Les micro-contrôleurs exploitent le plus souvent quelques kilo-octets de mémoire vive, et cette limitation de taille interdit le plus souvent le stockage de plusieurs configurations en mémoire de manière simultanée.
3. **Utilisation de la mémoire persistante** : L'utilisation de la mémoire persistante est nécessaire pour garantir que le processus d'adaptation assure des modifications transactionnelles, pour ainsi assurer le recouvrement de l'état du micro-contrôleur en cas de redémarrage sur erreur. L'EEPROM est une mémoire embarquée directement dans les micro-contrôleurs, le plus souvent avec une taille très limitée. Ce type de mémoire a une durée de vie limitée en terme de nombre d'opérations d'écriture. De manière similaire aux disques de type SSD [APW+08], les écritures dans une EEPROM doivent être distribuées sur les zones mémoires pour optimiser la durée de vie globale. L'utilisation de cette mémoire doit donc être limitée par la couche M@R pour assurer une pérennité du périphérique.
4. **Persistance d'état** : La capacité de recouvrement d'état est critique pour un système embarqué, qui est sujet à des pannes fréquentes (par exemple suite à une perte d'alimentation). Les micro-contrôleurs doivent alors redémarrer et restaurer la configuration précédente rapidement pour reprendre le fonctionnement nominal, et ceci en prenant en compte de nombreuses modifications d'architecture successives. Cette propriété est assurée par les périphériques car ils démarrent à partir d'une mémoire persistante. La mise à jour de leur micro-logiciel se fait *via* une opération de *flash* (écriture dans la mémoire) qui couvre ce besoin de persistance. L'adaptation dynamique doit donc respecter cette fonctionnalité.

D'une manière générale les CPS exploitent un large ensemble de nœuds autonomes avec de fortes contraintes énergétiques, ce qui incite à choisir des plates-formes peu

chères et capables de fonctionner sur de longues périodes sans acte de maintenance (par exemple pour changer une batterie). Les micro-contrôleurs de type AVR [1] prennent en charge ces besoins pour les raisons suivantes :

1. Leur base d'architecture 8 bits de type Harvard est simplifiée et robuste, rendant le temps d'exécution prévisible : un micro-contrôleur peut opérer dans une large bande de température (typiquement de -40 à 85 degrés Celsius), d'humidité, et d'alimentation électrique ; un tel micro-contrôleur a un nombre fixe de cycles pour exécuter une opération.
2. Leurs besoins énergétiques (et la chaleur dégagée) sont très faibles : un micro-contrôleur 8 bits fonctionnant à 32 kHz consomme typiquement 0,05 W (moins de 0.5 W à 1 MHz). Ils peuvent de fait fonctionner pendant de longues périodes sur batterie.
3. Leur architecture simplifiée permet la production de masse, faisant des micro-contrôleurs des nœuds très peu chers, capables d'être déployés en grand nombre en *cluster*.

Les micro-contrôleurs répondent donc aux besoins des CPS, ce qui explique, bien évidemment, leur usage intensif dans ce domaine et dans celui voisin des réseaux de capteurs. Si les AVR sont des processeurs relativement anciens au moment de cette évaluation, ils évoluent vers des architectures autour des produits de type ARM Cortex-M [2] plus puissants en tous points. Le choix des processeurs AVR pour cette évaluation correspond donc au pire cas, les résultats de viabilité seront de fait généralisables à de nouvelles architectures.

8.3 Capacité des micro-contrôleurs vis-à-vis des niveaux d'adaptation Kevoree

La modélisation des nœuds de type micro-contrôleur donne lieu à la création d'un *NodeType* Kevoree .

Les qualités de mémoire embarquée qui apporte la robustesse à ce type de noeud est également leur faiblesse. En effet toute modification de micro-logiciel embarqué nécessite l'écriture complète en mémoire *flash* du nouveau programme. C'est le cas par exemple lorsqu'on modifie une définition de type. Toute la difficulté est alors de trouver un compromis pour l'équivalence avec les concepts de Kevoree , entre la flexibilité offerte et le coût d'exploitation. Il faut alors trouver des solutions légères pour les quatre niveaux d'adaptation suivant leurs taux d'usage.

L'adaptation dynamique de *TypeDefinition* correspond à un ajout de fonctionnalité qui correspond par exemple à l'ajout d'un capteur physique. La mise à jour architecturale ou paramétrique correspond à un changement de configuration (par exemple un ajout de liaison) ou de contexte (par exemple un changement de paramètre). Les changements de *TypeDefinition* sont bien moins réguliers dans les cas d'usage car ils nécessitent

1. http ://www.atmel.com/Images/doc2545.pdf
2. http ://www.arm.com/products/processors/cortex-m/index.php

une intervention physique. La solution d'adaptation envisagée doit donc assurer un coût inférieur pour les adaptations architecturales, quitte à diminuer les performances des mises à jour de conception continue.

8.4 Implantation d'un nœud Arduino Kevoree

Le micro-contrôleur choisi pour l'expérience de validation est issu de l'environnement Arduino. Arduino [3] est un concept de plate-forme matérielle et logicielle *open source* pour la réalisation de prototype fondé sur un processeur AVR 8-bits. Connecté à des capteurs et actuateurs physiques ce type de plate-forme répond aux besoins des CPS et les résultats qui en découlent sont généralisables à d'autres familles de processeurs tels que les PIC ou les ARM.

L'implantation d'un tel nœud repose sur une équivalence entre les concepts Kevoree et les concepts manipulables dans ce type d'environnement. Ainsi les *TypeDefinition* reposent sur une implantation des composants en C. Le concept d'*Instance* de Kevoree repose sur une création dynamique de zone mémoire correspondant à ces structures tandis que les liaisons dynamiques entre *Composants* et *Channels* reposent sur des liaisons par tableaux de référence dynamique C. Les échanges de données suivent le modèle Kevoree : les ports sont implantés sous la forme de file d'attente de type FIFO, pour protéger les composants des accès externes. Les systèmes de base des micro-contrôleurs étant trop petits pour héberger une couche de système opérationnel (OS), aucun mécanisme d'ordonnancement n'est prévu. Pour pallier ce manque, un ordonnanceur est introduit pour gérer les files de messages et ainsi prioriser les instances. Cet ordonnanceur cherche à faire un équilibre entre les exécutions périodiques requises par certains composants et la taille des files d'attente. Il cherche ainsi à garder le système sain et éviter un phénomène de surcharge localisée.

Utiliser le flashage pour les évolutions majeures

Remplacer la totalité du micro-logiciel et redémarrer le périphérique est une implantation possible de l'adaptation dynamique pour le nœud micro-contrôleur. Si cette technique est raisonnable pour des périphériques connectés en filaire ou pour des maintenances physiquement connectées (opérateur physiquement présent) elle prend néanmoins plusieurs secondes. Cependant cette technique est problématique pour les périphériques non accessibles physiquement ou ayant des contraintes forçant une mise à jour en moins d'une seconde. Envoyer un micro-logiciel à travers les airs est une opération hasardeuse : en effet les micro-logiciels contiennent un grand nombre de données, et la gestion des erreurs de communication implique d'y ajouter un certain nombre d'informations supplémentaires pour valider la réception et gérer les erreurs. En pratique ceci allonge encore le temps de flashage au travers d'un réseau non filaire, rendant l'opération délicate et nécessitant un matériel et un micro-logiciel de démarrage dédié.

3. http ://www.arduino.cc

Dans l'approche proposée, le flash n'est requis que pour toute modification de *type définition*, par exemple lorsqu'un nouveau composant est ajouté à la librairie. De ce fait les micro-contrôleurs conçus en langage C n'offrent pas la même flexibilité que des environnements Java tels que les nœuds OSGi, car ils ne sont pas capables d'incorporation et de chargement de classe de type à chaud. Ceci est typiquement nécessaire pour une configuration initiale où les types envisagés sont déployés et pour les évolutions majeures du système (p. ex. pour prendre en compte un type non visible au déploiement initial). Pour tous les autres cas l'approche proposée permet de faire une reconfiguration des instances en mettant à jour uniquement une partie de la mémoire programme, rendant l'opération beaucoup plus rapide et garantissant l'encapsulation de l'approche Kevoree puisque le micro-contrôleur reste maître de sa plate-forme.

8.5 Validation expérimentale sur micro-contrôleur

Pour valider la viabilité de la solution M@R sur des environnements aussi contraints, une série d'évaluations a été réalisée sur des micro-contrôleurs réels.

8.5.1 Axes d'évaluation

Chacune de ces évaluations cherche à quantifier le surcoût sur chacun des axes détaillés dans les motivations. Ainsi l'évaluation globale porte sur les métriques suivantes :

- **Downtime** : temps d'adaptation globalement pris par le micro-contrôleur pour changer d'état et appliquer un nouveau modèle, incluant le temps d'envoi du nouveau modèle. Cette métrique définit le temps pris par le micro-contrôleur monotâche pour la gestion de son propre état et non pour la réalisation de sa tâche applicative (lecture de capteur par exemple). Ce délai est donc perdu d'un point de vue application métier.
- **Utilisation de la mémoire volatile (RAM)** : taux d'utilisation de la mémoire RAM dédiée à l'allocation dynamique d'instances Kevoree (composants, *channels*, etc). Cette métrique permet d'anticiper le nombre maximum d'instances qu'un nœud Arduino peut contenir.
- **Utilisation de la mémoire persistante** : taux d'utilisation de mémoire persistante pour stocker les états du nœud. Ce taux d'occupation dans le temps donne également de manière transitive une métrique qui évalue le taux de répartition du stockage et donc de l'usure de la mémoire. Par exemple la mémoire persistante embarquée dans les AVR 8 bits offre un nombre limité d'écriture certifié pour chaque octet et donc limite dans le temps le stockage possible de l'historique des états. Ce stockage est nécessaire pour la résilience de chaque reconfiguration.
- **Temps de redémarrage et de récupération** : temps pris par le micro-contrôleur pour restaurer son précédent modèle et son état après un crash, par exemple dû à une coupure de courant.

8.5.2 Protocole expérimental général

L'ensemble des expériences a été réalisé sur l'implantation de référence du noeud Kevoree pour Arduino. Le micro-contrôleur utilisé est un ATMEL AVR 328P. Ce processeur embarque 32 KB de mémoire flash pour stocker le micro-logiciel ainsi que 2 KB de mémoire RAM et 1 KB de mémoire persistante de type EEPROM. Une mémoire flash additionnelle (microSD connecté via un bus SPI) a été ajoutée pour certaines expériences afin de mesurer l'impact du type de mémoire sur les résultats.

Afin de simuler des changements de configuration un ensemble de modèles représentant les différents états sont créés. Ces modèles sont issus d'un cas d'étude de *smart building* tel que détaillé dans la publication à la conférence CBSE'12 [FBP+12]. De manière schématique ces modèles représentent un nœud micro-contrôleur qui pilote 5 capteurs physiques et qui peut communiquer avec un nœud passerelle à l'étage d'un bâtiment. Les modèles représentent différents usages de ce capteurs, un cas d'urgence où les capteurs échantillonnent beaucoup de valeurs et pilotent une alarme, un cas de confort où le capteur pilote une lumière et un chauffage, etc. Les différents modèles diffèrent d'environ 10 instances Kevoree , la figure 8.1 illustre un de ces modèles.

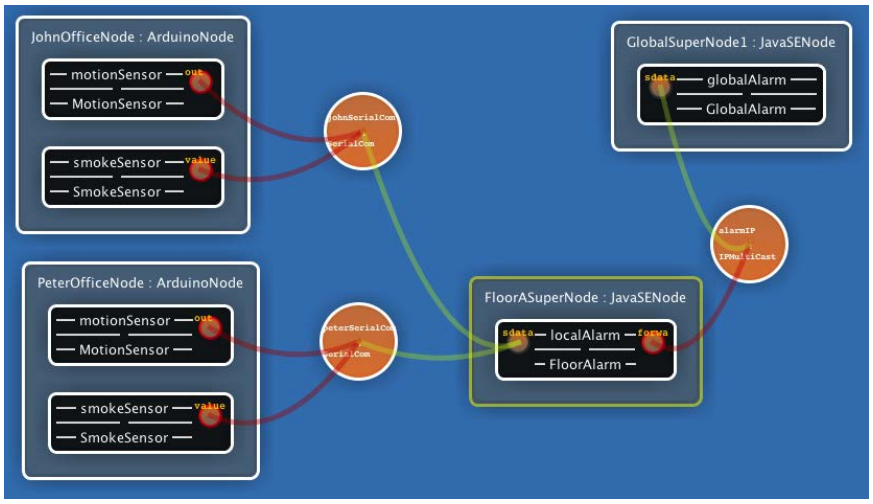


FIGURE 8.1 – Illustration modèle Kevoree de SmartBuidling

8.6 Downtime : combien de temps les adaptations bloquent-elles la logique métier ?

8.6.1 Configuration expérimentale

Dans cette expérience, 5 modèles tirés du cas d'étude *smart building* sont sélectionnés ; ils correspondent à des changements de configuration des capteurs d'une pièce pour un usage diurne ou nocturne. Dans ces modèles 4 nœuds sont présents, compre-

nant chacun de 0 à 10 *Instances* chacun. Chaque *Instance* utilise un *TypeDefinition* et comporte en moyenne environ 30 lignes de code.

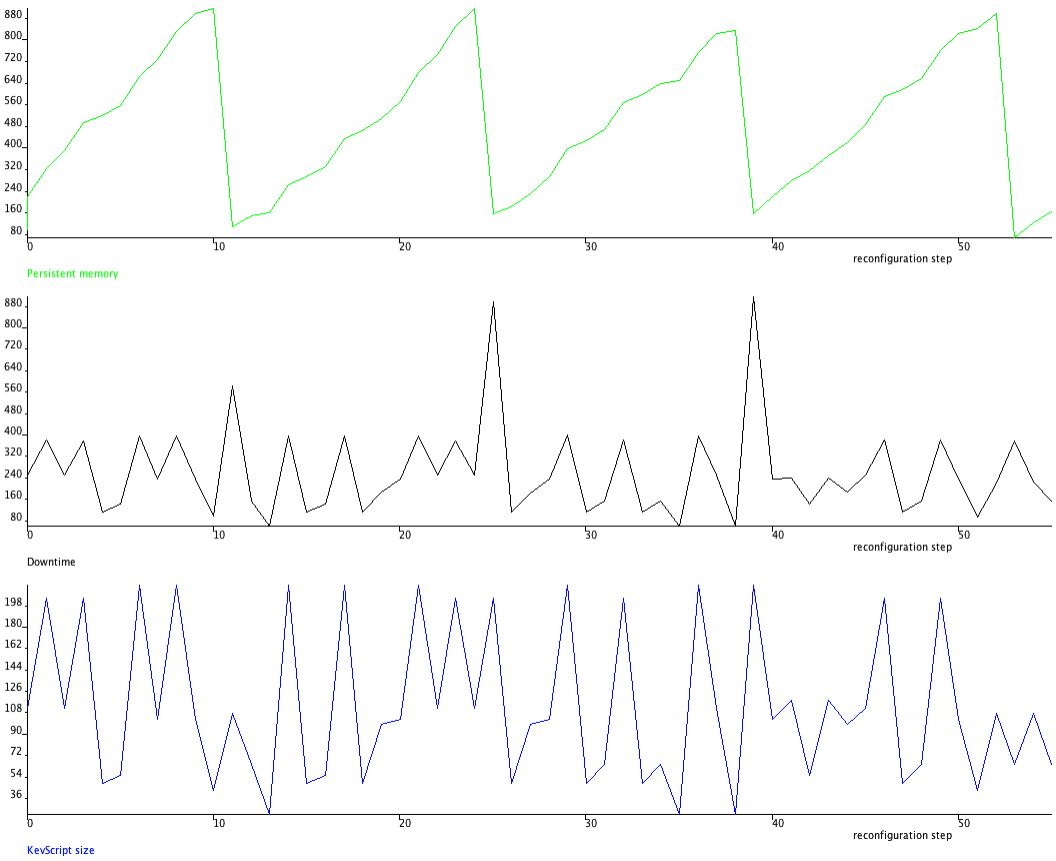


FIGURE 8.2 – Résultat expérimentaux bruts

Dans une première étape, le déploiement initial du premier état installe un modèle contenant tous les *TypeDefinition* exploités dans cette expérience. Cette mise à jour majeure est faite par une opération de flashage de la mémoire, précédée d'une étape de génération de code et compilation. Dans sa version expérimentale le nœud Kevoree Arduino introduit des sondes dans le code généré pour mesurer le temps d'inactivité (*downtime*) et l'usage de la mémoire (EEPROM et SDRAM). Après cette étape initiale, toutes les 100 ms un nouveau modèle est choisi de façon aléatoire et est synchronisé avec le micro-contrôleur, remplaçant ainsi la précédente configuration. Au total 500 changements d'état sont effectués consécutivement. La figure 8.2 représente les graphes des données brutes collectées lors de ce test. Le tracé du haut illustre l'usage de la RAM et démontre sa constance. Le second tracé illustre le temps de *downtime* par reconfiguration. Le troisième et dernier tracé illustre la taille de script de configuration pour piloter le nœud à distance.

8.6.2 Limites de validité expérimentale

8.6.2.1 Interne

La régularité des mises à jour de modèles (toutes les 100 ms) ainsi que le caractère synchrone du déploiement par le processus externe introduit un premier biais de validité interne. En effet ce déploiement synchrone régule les émissions de modèles en fonction de la capacité maximale de traitement du micro-contrôleur. Le délai de 100 ms est volontairement largement en dessous des valeurs nécessaires pour le cas d'usage sapeur-pompier, cependant pour des valeurs inférieures il serait alors nécessaire pour le nœud externe de faire tampon et de cumuler les mises à jour avant envoi au micro-contrôleur. Ce traitement introduirait alors un surcoût en temps qui serait alors dépendant de la capacité de calcul.

Le temps de prise de contrôle du micro-contrôleur est également dépendant de la rapidité d'interruption matérielle offerte par le support de transfert, ici une liaison série.

8.6.2.2 Externe

Le temps de déploiement initial pour la première configuration inclut un temps de compilation du micro-logiciel. Ce dernier est dépendant de la puissance du nœud de soutien connecté au micro-contrôleur.

Les délais présentés ici dépendent également de la vitesse de communication de la liaison entre le nœud de soutien et le micro-contrôleur. Réalisé ici avec une liaison série cadencée à 9600 baud, ceci représente une valeur inférieure à la capacité moyenne des puces de communications tel que Xbee mais dont la valeur varie en fonction de la distance géographique. De plus le canal expérimental filaire choisi prévient la plupart de erreurs de communication, le traitement de ces dernières inclut inévitablement un surcoût en temps pour l'émission et pour le micro-contrôleur pour valider la réception. Une extension de cette expérimentation est en cours de publication) pour l'évaluation précise de cette question.

8.6.3 Resultats et analyse expérimentale

Le déploiement initial et plus généralement les mises à jour majeures s'avèrent très coûteuses : le temps de *downtime* pour cette étape est de 12,208 s. Cette valeur importante s'explique notamment par le temps pris par le transfert du micro-logiciel mais également par le temps pris par le micro-contrôleur pour le redémarrage. Cette valeur varie dans une fourchette de +/- 2 secondes.

Les résultats de cette première expérience mettent en lumière la corrélation logique entre la taille des scripts de reconfiguration et les temps mesurés de *downtime* : le taux de corrélation de Spearman observé entre la taille des scripts et les temps de *downtime* est supérieur à 0,9. De plus, l'étape de compression exploitée pour réduire l'historique stocké dans l'EEPROMa également un impact sur le *downtime*. On observe alors que les déclenchements d'exécution de cette tâche sont directement corrélés avec les plus fortes valeurs de *downtime*.

Après 500 cycles de reconfiguration, on observe les valeurs maximales et moyennes suivantes :

- *downtime* minimum de 58 ms, 210 (c.-à-d. 12208 / 58) fois plus rapide que le flashage initial ;
- *downtime* maximum de 916 ms, 14 (c.-à-d. 12208 / 916) fois plus rapide que le flashage initial ;
- *downtime* moyen de 235 ms, 52 (c.-à-d. 12208 / 235) fois plus rapide que le flashage initial.

Voici la représentation en graphe et table de percentiles pour une meilleure analyse de la répartition des valeurs de *downtime*.

Percentile(%)	0	5	25	50	75	95	100
Downtime (ms)	58	59	139	221	248	398	916

La figure 8.3 synthétise le résultat de cette expérience et de la suite expliqué par la suite. Uniquement le premier graphe est donc ici exploité.

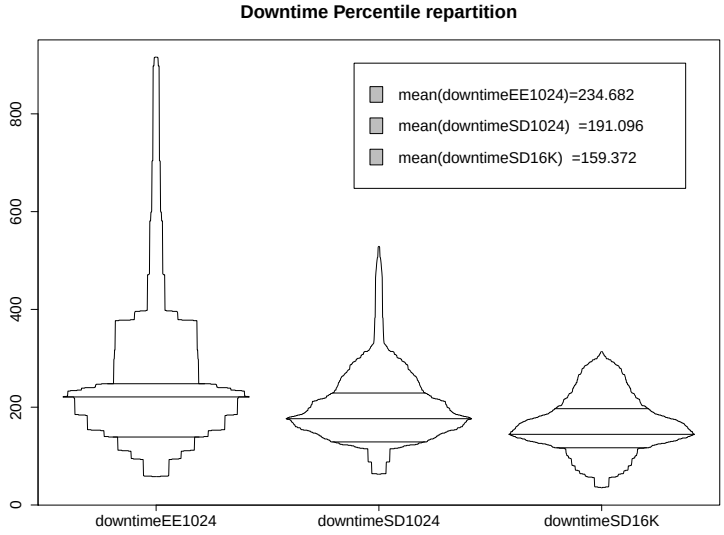


FIGURE 8.3 – Distribution percentile du temps de downtime(en ms)

La même figure 8.3 montre clairement que la répartition de valeurs de *downtime* se regroupe autour de 220 ms. 95% de ces valeurs sont inférieures à 400 ms et 75% sont inférieures à 250 ms. Seulement 5% de ces valeurs sont au dessus de la barre des 400 ms, ceci s'explique par l'étape de compression de l'EEPROM. La compression en mode juste à temps permet bien de limiter le nombre de pics de *downtime* et elle maintient les valeurs autour de 200 ms.

L'étude suivante de Tapani *et al* [mis] aborde justement l'identification d'une valeur seuil pour un périphérique fournissant un service à un humain. Cette étude conclut, je cite : « *Delays base on the human nervous system [...] It takes 190 - 215ms for light stimuli to be processed.*» La valeur seuil est donc particulièrement intéressante car elle correspond au temps de réaction perceptible par un humain. Une adaptation qui

reste proche ou en dessous de ce seuil sera perçue comme immédiate par un humain s'il en est l'origine. Les plus grandes valeurs de *downtime* sont systématiquement liées à une réduction de la mémoire EEPROM utilisée (routine de compression). On observe 32 compressions pendant les 500 configurations, soit 6.4% des reconfigurations nécessitant une compression pour être stockées. La valeur maximale de ces 32 pics est de 916 ms, la valeur minimale est de 218 ms et la moyenne est de 580,815 ms.

Les sondes injectées inspectent l'usage de la RAM pendant l'expérience, aucune fuite mémoire ni fragmentation n'est observée. Bien que le taux d'usage de la RAM soit stable il est nécessaire que le surcoût introduit par le *framework* d'instanciation dynamique ne limite pas la création d'un nombre réaliste d'instances. La prochaine expérimentation aborde ce point.

8.6.4 Extension expérimentale pour connaitre l'impact du type de mémoire.

Les mêmes 500 itérations ont été réalisées en remplaçant la mémoire EEPROM par une mémoire externe de type *flash* de 2 GB (carte SD) connectée *via* un bus SPI. Cette expérience est répétée deux fois, avec une taille de stockage de 1 kB (de manière identique à l'EEPROM) puis 16 kB. Les résultats sont montrés dans le tableau suivant :

Percentile(%)	0	5	25	50	75	95	100
Downtime SD 1K(ms)	63	88	129	176	229	324	529
Downtime SD 16K(ms)	35	56	117	145	197	297	314

La mémoire *flash* a un temps d'initialisation plus long que l'EEPROM, ce qui explique que la plus petite valeur de *downtime* de l'expérience soit plus grande que la plus grande de celle de l'EEPROM. Cependant la rapidité d'écriture et de transfert est plus grande, ce qui conduit à une homogénéisation des valeurs de *downtime* qui passe sous la barre des 200 ms dans la plupart des cas. On peut également noter qu'une valeur plus grande de mémoire persistante (16 kB) lisse les pics et fait légèrement baisser la valeur moyenne. Cependant la distribution des valeurs reste similaire. Les performances de la carte mémoire utilisée peuvent légèrement faire varier ces résultats.

8.7 Utilisation de la mémoire volatile : combien d'instance déployable ?

L'objectif de cette expérience est d'évaluer le surcoût en terme d'utilisation de mémoire volatile et ainsi valider la capacité restante vis-à-vis du déploiement des composants métiers.

8.7.1 Protocole expérimental

Pour mesurer cette capacité et donc le nombre maximal d'instances déployables, le protocole expérimental consiste à déployer une configuration initiale puis à l'étendre de façon cyclique. Cette configuration initiale du cas d'usage *smart building* comprend 3

instances : un composant *timer*, un composant gérant un interrupteur et un canal de communication entre les deux. Toutes les 100 ms ce modèle est étendu en ajoutant un nouveau composant interrupteur et en le liant au canal de communication existant. Des sondes sont injectées pour mesurer la mémoire SDRAM.

8.7.2 Limites de validité expérimentale

8.7.2.1 Interne

Le cas d'étude *SmartBuilding* exploite des composants de type capteur dont les tailles mémoires nécessaires pour leurs fonctionnement sont sensiblement identiques. Il en résulte une fragmentation de la mémoire du micro-contrôleur relativement faible. La fragmentation est inhérente à tous les processus d'allocation de mémoire lorsque les tailles de composants sont hétérogènes ; ceci n'est pas évalué ici.

8.7.2.2 Externe

Les résultats de cette expérience sont directement liés à la taille mémoire des composants choisis pour l'expérimentation. En effet la mémoire dynamique nécessaire pour chaque composant est elle-même liée à la complexité de son calcul de discrétisation de la valeur physique. Le type de capteur à mesurer influe donc sur les valeurs de ces résultats.

8.7.3 Résultats expérimentaux et analyse

L'expérience est menée jusqu'au remplissage total de la mémoire du capteur, interdisant tout nouveau déploiement. Deux types de micro-contrôleurs sont utilisés, un AVR ATMEL modèle 328P et un 2560, les résultats sont synthétisés dans le graphique suivant 8.4.

Les résultats pour le micro-contrôleur 328P montre que la mémoire SDRAM est pleine après environ 22 cycles de reconfiguration, ce qui veut dire que le micro-contrôleur a été capable de déployer 25 instances de composants (3 initiales et 22 additionelles). En pratique le nombre de périphériques géré par un micro-contrôleur est souvent proche du nombre de broches qu'il peut gérer. En supplément, un ou plusieurs composants additionnels sont nécessaires pour orchestrer les périphériques et faire les calculs dépendants. Dans notre cas on peut alors gérer 25 instances qui comprennent des composants d'orchestration et de gestion de périphérique comparativement aux 22 broches du micro-contrôleur de test.

Dans une deuxième expérience, un micro-contrôleur ATMEL 2560 est exploité avec 4 KB de mémoire. Les résultats convergent vers une valeur de 100 instances déployées, l'amélioration de 300% est donc cohérente avec l'augmentation de capacité. Encore une fois le nombre d'instances est plus grand que le nombre de broches (80) que le micro-contrôleur peut gérer. Malgré un surcoût mémoire, l'approche est donc compatible avec l'état de la pratique.

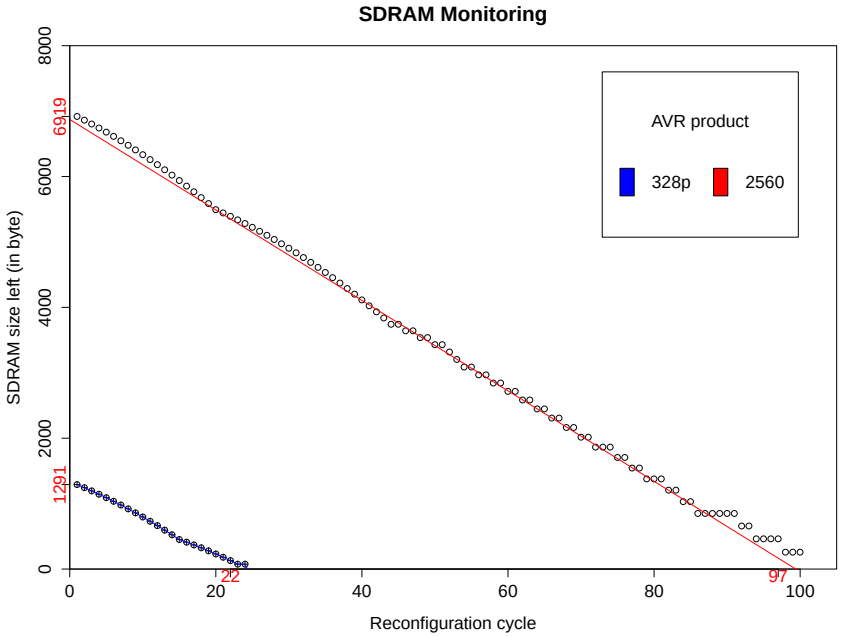


FIGURE 8.4 – Résultat expérimental sur la capacité mémoire volatile

8.8 Utilisation de la mémoire persistante : combien de reconfigurations successives peuvent être déployées ?

Les micro-contrôleurs ont des capacités limitées en terme d'écriture. Ces contraintes venues du matériel exploité dans le monde de l'embarqué peuvent être antinomiques avec l'usage de l'historique nécessaire pour la persistance du Model@Runtime. L'expérience suivante vise donc à valider cet usage dans le cas d'usage *smart building*. Le protocole expérimental est le même que pour la première expérience 8.6. Cinq modèles du cas d'usage standard sont déployés de manière cyclique. Des sondes sont injectées afin de mesurer les écritures dans la mémoire persistante. Dans cette première expérience, la mémoire embarquée EEPROM du micro-contrôleur 328P est utilisée, 500 cycles de déploiement sont effectués.

8.8.1 Limites de validité expérimentale

8.8.1.1 Interne

Les écarts types (du nombre d'instances modifiées) entre chaque configuration influent sur le nombre total de configurations à sauver sur le support persistant.

8.8.1.2 Externe

Les résultats de cette expérience sont directement liés à la taille mémoire persistante nécessaire pour le stockage d'un composant. Cette taille par composant est elle-même

directement liée aux nombres et aux types de paramètres dynamiques de celui-ci. Les composants choisis pour l'expérience sont représentatifs du cas d'étude et comprennent entre 1 et 4 paramètres de type entier.

8.8.2 Résultats et analyse

On observe 32 pics de compression de la mémoire EEPROM durant les 500 cycles soit en moyenne une compression tous les 15,625 changements de modèle. De manière analogue à la mémoire employée dans les *Solid State Disks* [APW+08], chaque opération d'écriture sur la mémoire EEPROM entraîne une usure. Cette usure doit être répartie de manière uniforme sur la mémoire pour éviter la perte prématurée de zone mémoire. Dans le pire des cas l'algorithme de compression effectue un cycle d'écriture sur chaque octet de la mémoire pour la compression d'un état initial. Chaque octet de ce type de mémoire est garanti pour 100 000 écritures [4]. Par extrapolation si on suppose que 100 reconfigurations peuvent être effectuées par jour (ce qui est bien plus que le cas d'usage envisagé), chaque octet de l'EEPROM sera écrit 6,4 fois par jour en moyenne. Dans ce cas d'usage la mémoire mettra 15 625 jours à atteindre son taux d'usage certifié, soit une durée de vie d'environ 43 années pour ce système piloté par le Model@Runtime.

8.9 Délai de redémarrage : combien de temps pour la récupération d'état ?

La sauvegarde d'état et sa restauration prennent un temps non négligeable lors du démarrage du capteur. La persistance de cet état est un besoin critique dans ce cas d'usage, car le capteur est souvent confronté à des pertes d'alimentation électrique. L'expérience cherche à montrer que la couche Models@Runtime permet un démarrage compatible avec les besoins du cas d'usage.

Le protocole expérimental général est ici réexploité sur 50 cycles de reconfiguration espacés de 2 secondes. Le micro-contrôleur est physiquement redémarré entre chaque reconfiguration. Une nouvelle sonde est ajoutée dans le micro-logiciel pour la mesure du temps de démarrage de la couche d'adaptation générée.

8.9.1 Limites de validité expérimentale

8.9.1.1 Interne

Le *framework* exploité (librairie Arduino) pour l'implantation de Kevoree peut avoir une influence négative sur les performances de lecture sur les mémoires externes.

8.9.1.2 Externe

Les performances en lecture de la carte SD exploitée pour l'expérience peuvent influencer légèrement les résultats. Les performances de la carte utilisée (de type 4)

4. http ://arduino.cc/en/Reference/EEPROMWrite

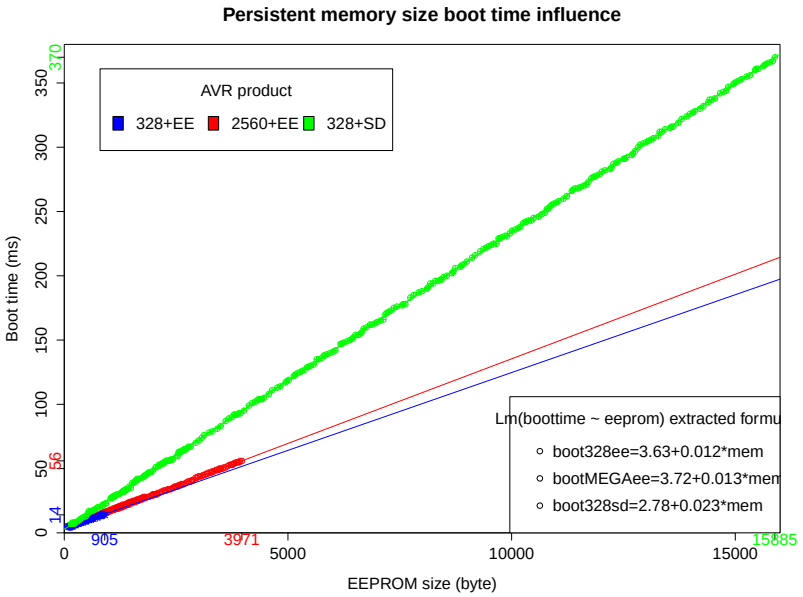


FIGURE 8.5 – Influence de la mémoire persistante sur le temps de démarrage

correspondent cependant à la moyenne des performances de ce type de mémoire.

8.9.2 Résultats expérimentaux et analyse

Les premières mesures sont effectuées sur un micro-contrôleur 328P et une mémoire interne EEPROM. Ils sont représentés dans la figure 8.5 par la courbe verte (les autres tracés de la figure font partie de la deuxième série de mesures expliquée par la suite).

Ces résultats montrent que le pire cas avec ce processeur AVR et lorsque la mémoire EEPROM de 1 KB est pleine entraîne un temps de démarrage d'environ 15 ms. Dans les meilleurs mesures, lorsque la mémoire est relativement vide après une compression d'état, le temps de démarrage devient alors de 3 à 4 ms. Cette valeur seuil proche du pire cas est notamment explicable par la lenteur du lecteur de ce type de mémoire. Cependant le temps de démarrage reste largement acceptable dans ce cas d'usage car inférieur aux 200 ms décrits dans l'étude de Tapani *et al* [mis] perceptible pour un humain même dans le pire des cas. Il est même possible d'inférer que le temps de démarrage étant beaucoup plus court que le temps d'écriture, il est intéressant d'exploiter l'ensemble de la mémoire disponible.

Extension du protocole avec un nouveau type de mémoire

Une deuxième série de mesures est lancée en remplaçant la mémoire EEPROM par une mémoire de type *flash* de taille de 1 KB et 16 KB. Cette mémoire externe de type SD, connectée par un bus SPI entraîne alors un temps de démarrage plus long. On observe un coefficient de 0.012 pour l'EEPROM comparativement à un coefficient de 0.023 pour la mémoire SD. Ceci s'explique par les temps d'écriture et lecture ralentis

par le bus. Le temps de démarrage est linéairement distribué en fonction de la taille allouée (1 ou 16 KB). Au-delà de cette limite le temps de démarrage (360 ms) dépasse la valeur moyenne du temps de reconfiguration moyen et perd alors son intérêt.

8.10 Comparatif vis-à-vis d'un micro-logiciel non généré

Une dernière expérience permet de quantifier le surcoût de l'approche dynamique par rapport à un micro-logiciel statique écrit à la main par un humain. Pour cette quantification on mesure la taille mémoire volatile et programme pour un exemple *HelloWorld* en C écrit à la main et un autre généré. Les micro-logiciels résultants sont déployés sur un micro-contrôleur 328P. Après la séquence de démarrage la version C laisse 1842 octets disponibles, tandis que la version générée par Kevoree avec le *framework* dynamique laisse 1604 octets libres. Le surcoût représente donc 11% de la mémoire RAM total disponible (2048). La version C prend également 2.3 KB de mémoire programme et la version Kevoree 7.3 KB soit 15% des 32 KB disponibles. Le surcoût sur la plus petite puce de cette étude (et donc le cas le plus défavorable), se limite donc à moins de 15% des différentes mémoires disponibles , bien sûr ce taux diminue sur des puces plus puissantes.

8.11 Conclusion vis-à-vis des axes d'évaluation

Sur chacun des axes qui ont fait l'objet d'une expérimentation, l'introduction de la couche d'adaptation dynamique dirigée par le Model@Runtime s'est avéré compatible avec les pré-requis de l'embarqué et du cas d'usage. Bien évidement les seuils d'acceptabilité de temps d'arrêt ou de redémarrage sont à mettre en corrélation avec les cas d'usage. Ainsi si l'approche exploite ici un cas d'usage issu de l'informatique embarquée pour cette validation, celle-ci ne couvre néanmoins pas tous les cas d'usage critiques. Certains cas d'usage pourraient par exemple nécessiter des valeurs seuil inférieures aux résultats, cependant l'approche d'évaluation aux limites valide ici un cas d'usage fortement lié à l'Internet des Objets. Ce cas limite est donc compatible d'un point de vue logique et technique avec l'approche Model@Runtime, rendant possible l'intégration des CPS dans la modélisation d'un DDAS global.

Chapitre 9

Évaluation quantitative aux limites : adaptation de DDAS en environnement mobile

Outre la gestion de l'hétérogénéité et des différents environnements d'adaptation, l'autre axe d'évaluation critique de Kevoree vis-à-vis du cas concret sapeur-pompier est la capacité de synchronisation des nœuds. En effet du cas d'étude DAUM se dégage la problématique difficile de la synchronisation des nœuds mobiles pouvant chacun à tout moment décider d'une adaptation.

Ce cas d'usage aux limites cette fois dans la complexité en largeur due aux nombres de nœuds à coordonner, met à l'épreuve l'abstraction Kevoree sur deux points :

- sa capacité à encapsuler un algorithme de convergence afin de faire converger les copies de modèles des différents nœuds ;
- sa capacité à encapsuler un algorithme de dissémination qui permet la dissémination dans un *cluster* large échelle.

Cette section traite de l'évaluation de l'implantation du *GroupType* Kevoree alliant une inondation *gossip* et une traçabilité par *VectorClock* détaillée en 6.4.7 vis-à-vis de ces besoins. En effet l'inondation par *gossip* est justement choisie pour résister à un réseau maillé à la topologie très dynamique tandis que les *VectorClock* permettent la traçabilité des modifications. Par cette traçabilité on peut alors déterminer l'existence de sous-groupes isolés déterminant seuls de nouvelles configurations et divergeant du modèle commun. L'approche est donc nettement inverse des groupes consensus, puisqu'ici le but est de laisser les nœuds diverger pour résister à la perte de connexion et continuer d'offrir un service sur le terrain. La validation présentée ci-dessous exploite donc une implantation en Scala et Java des concepts détaillés en 6.4.7 sur un *cluster* de machines servant à simuler l'environnement mobile sapeur-pompier.

9.1 Validation expérimentale sur *cluster* de simulation

Une validation qualitative et quantitative a été effectuée sur cette implantation d'algorithme issue des travaux du monde de l'informatique distribuée, selon les indicateurs suivants :

- mesure du temps de propagation d'un nouveau modèle vers des nœuds membres d'un groupe ;
- capacité à résister à la perte de connexion entre nœuds ;
- capacité à détecter des modèles issus de modifications et donc de branches concurrentes et capacité à les réconcilier.

Pour chacun de ces indicateurs, un protocole expérimental a été mis en place pour simuler sur une grille de calcul les comportements d'un système vis-à-vis des reconfigurations envisagées dans le cas sapeur-pompier. Bien que la cible de production soit un environnement embarqué sur tablette de type Android, l'usage de la grille permet ici une simulation plus précise et à plus large échelle du comportement du réseau. Cependant le biais introduit ici ne permet de valider que le comportement de l'algorithme et non le temps de réponse absolu nécessaire pour le déploiement sur le terrain.

9.2 Protocole expérimental commun

Toutes les expériences suivantes partagent un protocole expérimental commun. Chacune utilise un ensemble de nœuds logiques Kevoree déployés sur un nœud plate-forme physique de la grille de calcul. Chaque nœud logique exploite donc sa propre machine virtuelle Java sur l'implémentation du nœud JavaSE de Kevoree . La grille expérimentale exploitée pour ces expériences est composée de nœuds de calcul hétérogènes en terme de type et de puissance. Les communications sont supportées par un réseau local à 100 MB/s.

9.2.1 Modèle de topologie initiale

Chaque expérience prend en entrée un modèle de démarrage qui décrit l'architecture de plate-forme de façon abstraite. Ce modèle décrit principalement la topologie des nœuds, leurs liens de communication disponibles ainsi que les instances du groupe sous test. Ce même modèle est exploité par les implantations de groupe de synchronisation pour débuter les communications de fragment à fragment. En construisant des modèles de topologie on peut alors influencer les communications des groupes et ainsi simuler des comportements vis-à-vis de topologies rencontrées dans le cas sapeur-pompier. On peut par exemple simuler des communications indirectes en supprimant un lien entre un nœud A et B, pour l'obliger à passer par un nœud tiers C. Ce modèle de topologie est donc clairement utile pour la simulation mais introduit un biais dans l'expérience, car on ne peut observer la construction de la topologie par le groupe lui-même. Ainsi dans un cas réel cette approche serait à remplacer par une approche similaire à T-Man de Jelasity *et al* [JB06a]. Les résultats de construction de topologie ne seront donc pas analysés ici en raison de ce biais.

9.2.2 Horloge de temps absolu pour la collecte des traces d'exécution

Afin de tracer la propagation des modèles et leur application dans le système, l'algorithme et l'implantation Java du groupe Gossip+VectorClock ont été décorés avec un *logger* en temps absolu. Celui-ci envoie des traces à chaque changement d'état interne, décrivant le groupe, le *VectorClock* courant et l'identification de la plate-forme origine de la propagation de modèle, ainsi que des métriques d'usage réseau. Afin d'exploiter les données temporelles de ces traces la synchronisation des traces effectue la réconciliation du temps d'émission avec une horloge de référence (en utilisant Java Greg Logger [1]). De façon plus précise, la synchronisation est fondée sur une architecture client/serveur, chaque client synchronisant alors de façon périodique sa latence observée avec le serveur référent. Ainsi chaque envoi peut réconcilier le temps absolu en prenant en compte cette latence observée plus la latence réseau. Puis les traces sont chainées en observant les *VectorClock* récoltés et ainsi en ordonnant suivant l'ordre d'apparition des différentes versions prises par les modèles d'architecture. En résultat on dispose d'une liste de traces d'exécution retraçant les états temporisés des nœuds du cluster.

9.2.3 Mode de communication

Deux *patterns* d'échange sont principalement exploités pour construire l'algorithme. Le terme **période de pooling** est associé au temps passé entre deux synchronisations actives, initiées par un membre du groupe vers un autre. Dans cette phase de synchronisation un *VectorClock* ou un modèle est envoyé au membre origine.
La technique du **push/pull** est l'association d'une communication périodique et d'une communication par évènements. Ce mode permet l'envoi de notification à tous les groupes accessibles lorsqu'un nouveau modèle est prêt.

9.3 Études expérimentales

Trois expériences évaluent l'implantation de Kevoree et de son *GroupeType* Gossip+VectorClock. Ainsi on retrouve détaillées ci-dessous l'expérience #1 visant à l'évaluation des temps de dissémination du modèles, l'expérience #2 évaluant la résistance aux fautes et l'expérience #3 évaluant la résistance à l'apparition de branches divergentes et la capacité de réconciliation de modèles.

9.3.1 Délai de propagation vis-à-vis de l'usage du réseau de communication

Cette première expérience cherche à mesurer de façon précise la capacité du *GroupType* à disséminer des modèles sur une topologie maillée. Cette mesure permet également de comprendre l'impact du délai de propagation vis-à-vis de l'usage de la bande passante réseau.

1. `http://code.google.com/p/greg/`

9.3.1.1 Protocole expérimental

Comme décrit dans la section du protocole expérimental commun, les mesures sont effectuées sur une grille dans un environnement équipé de sondes et contrôlé par un modèle de topologie inclus dans un modèle Kevoree . Après l'étape de démarrage sur ce modèle, un nœud est choisi périodiquement de façon aléatoire pour injecter un nouveau modèle dans le réseau. En pratique ceci se traduit par le calcul d'un nouveau modèle (par une machine extérieure pilotant le test) en simulant une migration d'une instance de composant d'un nœud à un autre (cas d'adaptation élastique). Ce nouveau modèle est réinjecté dans le nœud cible et le groupe instance est alors automatiquement notifié pour réaliser la propagation. Cette nouvelle configuration possède un tatouage qui permet la traçabilité de la source du modèle. La figure 9.1 montre le modèle de topologie utilisé pour cette expérience dédiée à la communication multi-saut (*multi-hop*).

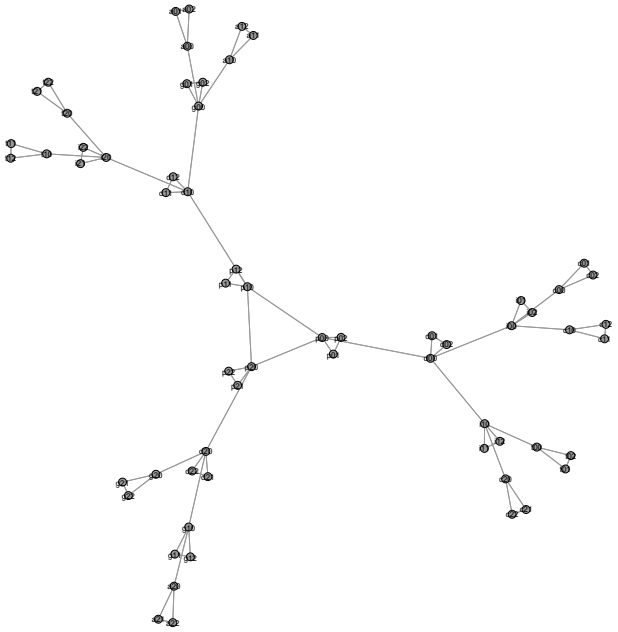


FIGURE 9.1 – Modèle de topologie, expérience #1

Dans cette configuration statique 66 nœuds composent le réseau, et aucun n'est ajouté ni retiré durant l'expérience. Le modèle de topologie réseau influence directement et de manière significative les résultats, sa définition doit donc mettre en lumière les éléments à valider dans cette expérience dédiée au délai de propagation. Ainsi suivant les descriptions de Shudong *et al.* [JB06b] les dérivations de modèle *SmallWorld* en réduisant les liens directs permettent de simuler la communication multisaut et l'agrégation de nœuds, comme dans le cas du modèle utilisé en 9.1

Si le modèle de topologie est fixe l'expérience fait varier les paramètres suivants : délai de temps de *pooling* et changement du mode de communication *push/pull* ou uni-

quement *pull*. Deux lancements distincts permettent de tester les deux cas limites de l'algorithme, une première configuration sans les notifications avec un délai de synchronisation active de 1000 ms, et une deuxième avec activation et un délai de 15 s. Dans les deux cas, une reconfiguration est injectée toutes les 20 secondes, et 12 reconfigurations sont effectuées successivement.

9.3.1.2 Limites de validité expérimentale

Interne Le surcoût introduit par les communications réseau lors de l'envoi de notifications est ici évalué dans son pire cas car réalisé par le biais d'un envoi multiple. Les sondes mesurent la quantité de données envoyées et reçues par les nœuds. Or dans le cas de réseau de type IP des mécanismes tels que le *multicast* permettent de réduire la recopie des données sur le réseau. Par extension ce type d'amélioration serait également disponible sur des technologies de réseau de type radio.

Externe De par le caractère homogène du réseau interconnectant la grille de calcul, l'hétérogénéité de vitesse de transport n'est ici pas évaluée. Cependant l'hétérogénéité des nœuds de calcul introduit de fait une différence de temps de traitement qui permet d'observer et d'évaluer l'algorithme sur des temps de propagation variables.

9.3.1.3 Analyse des résultats expérimentaux

Les propagations mesurées par sauts sont présentées sous la forme d'un graphique indiquant leurs distributions percentiles 9.2.

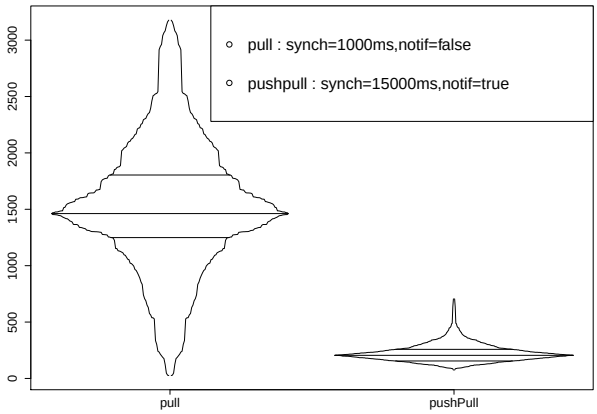


FIGURE 9.2 – Delai/hop(ms)

Les valeurs représentées sont les valeurs brutes après réconciliation par l'horloge absolue divisées par le nombre de sauts entre la cible et l'origine du nouveau modèle émis (calculé suivant un algorithme de Bellman-Ford [CRKGLA89]). Le trafic réseau en volume de messages de protocole est représenté sur la figure 9.3 en KB par nœud et par

reconfiguration. Ce volume n'inclut pas la charge utile. Les valeurs absolues de consommation réseau dépendent beaucoup de l'implantation du *GroupType* . Les résultats présentés ici sont inhérents à l'implantation Java et ils seraient bien évidemment différents pour l'implantation Android ou sur microcontrôleur. L'utilisation de notifications réduit significativement le délai de propagation moyen : celui-ci passe de 1510 ms/saut à 215 ms/saut. De plus, la représentation percentile montre clairement que l'écart-type propagation diminue avec l'emploi de notifications. En considérant uniquement le délai et non la charge réseau induite par l'inondation de messages, la capacité de passage à l'échelle de cette version est bien meilleure sur de grands *clusters*. Cependant en comparant cette version *push/pull* avec le *pull* simple, l'utilisation de notifications n'a pas un impact significatif sur le réseau. L'analyse plus fine des résultats a également permis d'associer des variations de cet impact suivant les types de cycle dans la topologie. En effet, la vitesse de propagation provoque la création de branches concurrentes au sein de sous-*clusters* puis leur diffusion. Ceci augmente le nombre de conflits à résoudre, même si l'algorithme réseau de dissémination est en revanche plus performant. Lorsque les notifications sont désactivées, les délais sont suffisamment longs pour éviter ces créations de branches concurrents inutiles ; ceci supprime alors des transports réseaux non nécessaires et l'inondation par des modèles concurrents. Comme la charge utile de cet algorithme contient lui-même des informations de topologie (modèle Kevoree) il serait possible d'optimiser ces créations de branches en tenant compte de ces informations pour prévenir cette inondation. En résumé, en termes de délai de propagation le gain d'ajout de notification introduit un surcoût dans les communications réseau. Ce surcoût reste acceptable et peut être traité avec une extension de l'algorithme.

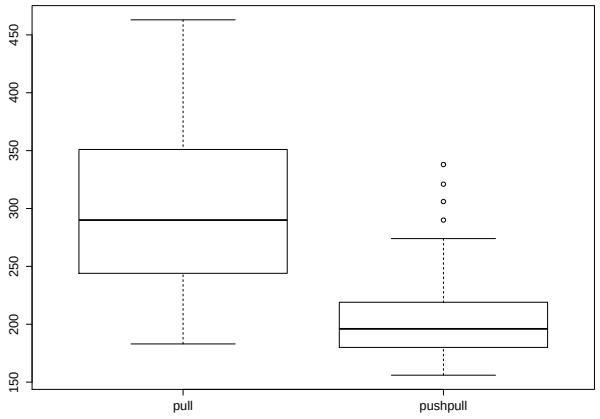


FIGURE 9.3 – Utilisation réseau/nœud(in kbytes)

9.3.2 Impact des erreurs de communication sur les délais de propagation

Un réseau mobile de terrain tel que celui exploité pour le système d'information tactique du cas sapeur-pompier est caractérisé par un un grand nombre de nœuds devenant fréquemment inaccessibles. L'algorithme à évaluer est justement défini pour résister à ces topologies à problèmes. Cette deuxième série de mesures évalue la capacité du groupe à disséminer un modèle dans un réseau possédant un fort taux de défaillance des liens entre nœuds.

9.3.2.1 Protocole expérimental

Ce protocole reprend les grandes lignes du précédent, le modèle de topologie est cependant remanié pour simuler un réseau maillé avec de nombreuses routes alternatives entre les nœuds (voir la figure 9.4).

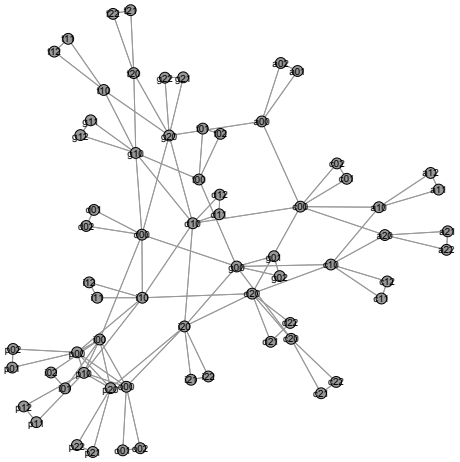


FIGURE 9.4 – Topologie pour expérience #2

De manière similaire à l'étape précédente un nouveau modèle représentant une migration de composant est injecté périodiquement. Durant chaque lancement de modèle, de nouvelles fautes de réseaux sont injectées entre les nœuds, selon une distribution de Poisson. Ainsi le taux d'erreur du réseau augmente à chaque modèle calculé, et le nombre de nœuds accessibles diminue.

Pour effectuer les simulations d'erreurs, des sondes sont injectées dans les machines virtuelles pour manipuler les cartes réseaux, ces sondes surveillent et synchronisent les évènements de synchronisation envoyés à chaque exécution du protocole du groupe. À chaque modèle injecté une liste de nœuds théoriquement accessibles est calculée, on peut alors comparer cette liste avec les évènements réellement reçus par les autres nœuds pour calculer le temps de propagation moyen et vérifier la propagation globale.

9.3.2.2 Limites de validité expérimentale

Interne De manière analogue à l'expérience précédente, l'homogénéité du réseau ne permet pas d'observer ici l'impact de temps de transport entre les nœuds de communication. Cependant le mode de propagation de type *gossip* employé ici n'exploite pas les débits les choix de nœuds à synchroniser, ce qui limite l'impact de ce biais.

Externe Les fautes sont injectées par le biais d'une désactivation du réseau, ce qui correspond à une panne totale de communication ou de noeud de calcul, l'évaluation d'une panne intermittente (avec un taux de pertes très élevé par exemple) n'est pas réalisée ici.

9.3.2.3 Analyse des résultats expérimentaux

La figure 9.5 illustre les résultats de l'expérience #2. L'histogramme qui y est représenté synthétise le taux de pannes réseau simulées à chaque lancement. La courbe rouge illustre le temps moyen de propagation (en millisecondes) à tous les noeuds atteignables. Au delà d'un taux de panne dans le réseau supérieur à 85%, le noeud origine de la nouvelle configuration est isolé du réseau et ceci termine alors l'expérience. En dessous de ce seuil, l'expérience montre que tous les noeuds atteignables reçoivent le nouveau modèle. Entre 0% et 60% le temps de propagation est variable dans une fourchette de 600 à 800 ms. Malgré ces variations dues essentiellement à la topologie et donc un nombre de sauts variables entre deux points, le taux de panne n'a pas d'impact décroissant sur le temps de dissémination.

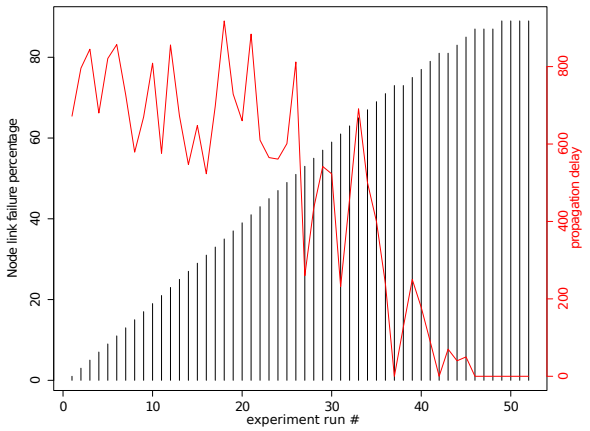


FIGURE 9.5 – Resultats expérience #2

9.3.3 Réconciliation de modèle et reconfiguration concurrente

Cette dernière expérience évalue la capacité du groupe à détecter et réconcilier les modèles émis de façon concurrente par les nœuds. Ce type de problème arrive par

exemple dans le cas sapeur-pompier en partie à cause des communications sporadiques dans le réseau. Un nœud ou un groupe de nœuds peut alors être isolé pendant une période. En conséquence les reconfigurations distantes ne sont pas appliquées, de plus, de nouveaux modèles peuvent être calculés localement et diffusés dans le sous-réseau. Des reconfigurations concurrentes apparaissent alors au moment du rétablissement de la connexion . L'implantation du groupe se fonde alors sur la traçabilité des *VectorClock* pour détecter ces conflits directement sur les modèles émis. Notre expérience étudie cette réconciliation dans ce cas d'usage.

9.3.3.1 Protocole expérimental

Le protocole de l'étape précédente est simplifié pour exploiter uniquement une topologie de 12 nœuds, comme illustré par la figure suivante 9.6 :

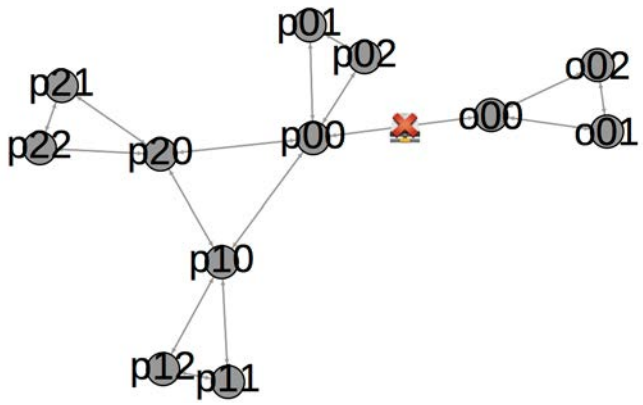


FIGURE 9.6 – Topologie pour l'expérience #3

Une configuration initiale *c1* est injectée dans un nœud *p00* juste après la phase de démarrage. Toutes les sondes mettent alors les liens réseaux en position valide et le modèle est alors propagé à l'ensemble des nœuds. Une faute est par la suite injectée entre les nœuds *p00* et *o00* et illustrée par une marque rouge sur le schéma. Les nœuds *o00*, *o01*, *o02* se retrouvent alors isolés. Un nouveau modèle est ensuite injecté au nœud *p00* (c2) et un modèle différent au nœud *o00* (c3). Un délai de 1000 ms sépare chaque - injection et l'algorithme est configuré avec des notifications et une période de *pooling* de 2000 ms.

9.3.3.2 Limites de validité expérimentale

Lors de la détection d'un conflit le nœud local doit alors faire une résolution qui correspond à l'assemblage des modifications stockées dans deux modèles différents, qu'il faut fusionner à l'aide de différentes stratégies, par exemple avec des règles de priorités. Le temps de calcul de cette résolution est considéré comme négligeable et non pris en

compte ici, mais cependant suivant les stratégies de résolution ce temps peut s'avérer plus coûteux sur un très large réseau.

9.3.3.3 Analyse des résultats expérimentaux

La figure 9.7 illustre les résultats de cette troisième expérience. Ceux-ci sont directement dérivés de notre résultat d'arbres de traces.

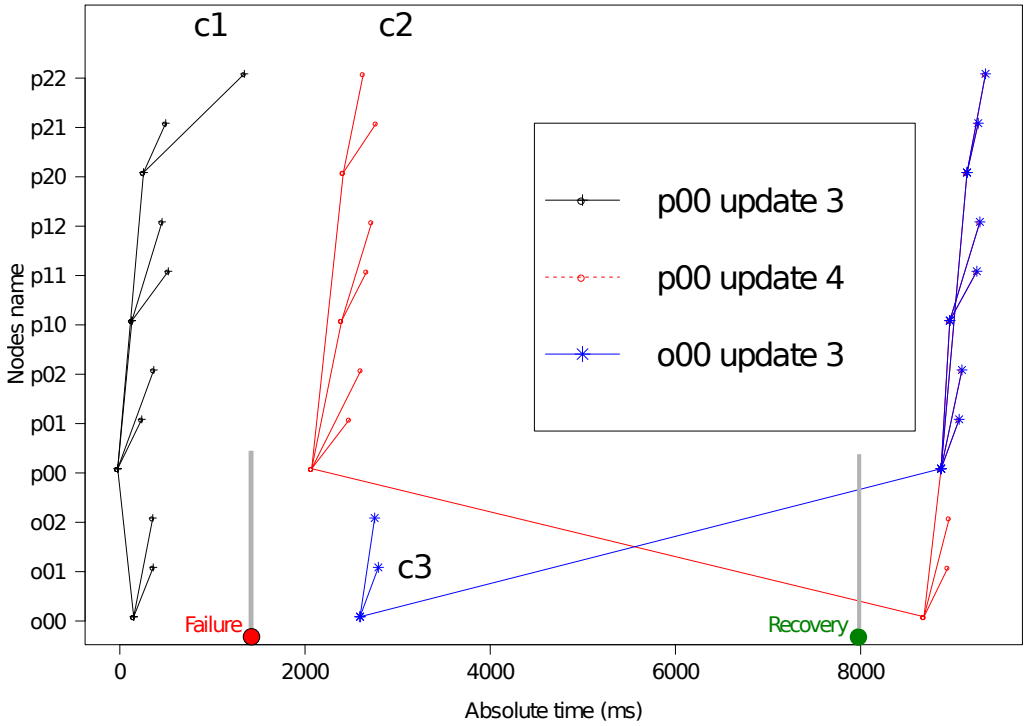


FIGURE 9.7 – Réconciliation de modèle émis en concurrence

Trois propagations de modèle sont représentées comme une succession de lignes qui représentent chacune la propagation d'un modèle d'un nœud à un autre. Le premier modèle envoyé sur le réseau sain et représenté en noir au temps 0 ms. Le troisième modèle injecté dans le nœud *o00* (au temps 2500ms) est représenté par la trace bleue tandis que le dernier injecté dans le nœud *p00* (time 2000ms) est illustré en rouge. Le premier modèle atteint directement tous les nœuds. Au temps 1500 ms la perte de connexion est alors simulée, le second modèle injecté en *p00* atteint tous les nœuds sauf ceux joignables uniquement *via o00*. De manière symétrique le second modèle qui est injecté dans le nœud *o00* n'est pas propagé aux nœuds après le nœud *p00*. Au temps 8000 ms, la simulation d'erreur de communication commencée au temps 1500 ms est annulée. Après un délai de synchronisation de 380 ms on observe la réconciliation du modèle et sa propagation dans sa forme fusionnée à tous les nœuds *via* la trace représentée en violet.

9.4 Conclusion sur l'usage des groupes pour la convergence

A défaut de pouvoir proposer une évaluation exhaustive des implantations possibles des *GroupType* Kevoree dans les domaines des algorithmes de dissémination distribués, cette section a présenté un cas limite mettant en lumière les capacités de divergences et convergences de Kevoree . L'aspect exhaustif et l'état de l'art sur l'écosystème Kevoree sera abordé plus en détails dans la section suivante permettant ainsi d'évaluer la maturité de l'évaluation de l'abstraction vis-vis d'autres implantations. Cette série d'expériences permet cependant de mettre en lumière l'adéquation des résultats obtenus avec les besoins du cas sapeur-pompier. En effet que ce soit sur les délais réseau ou la capacité de détection et correction des modèles concurrents l'implantation résiste correctement aux paliers demandés. De manière plus générale ceci valide également l'hypothèse que le surcoût introduit par un style de programmation opportuniste laissant diverger le système est envisageable, et peut largement s'appuyer sur les résultats du distribué pour être amélioré et exploiter des heuristiques adaptées à chaque cas d'usage. Enfin ceci met là encore en lumière le besoin d'exprimer la diversité des algorithmes de convergence qui par définition existe de part ces variantes.

Chapitre 10

Ecosystème Kevoree et validation par les projets liés

Ce dernier axe de validation cherche à démontrer l'utilisabilité (section 10.1) et la versatilité (section 10.2) du modèle proposé par la maturité de son prototype et son usage dans différents projets liés, tels que le projet DAUM ou encore l'usage pour la modélisation et l'adaptation de système de type Cloud computing. Le projet Kevoree a donné lieu à l'implantation d'un méta-modèle qui spécifie la structure du modèle d'architecture et s'est accompagné de plusieurs projets et DSL liés afin d'outiller le développement des composants, ceci est détaillé en sous-section 10.1.1 et 10.1.2. En reprenant les outils dédiés à la modélisation avant le déploiement de systèmes et les outils des architectures à composants existantes, Kevoree donne lieu à un environnement de développement qui a dû revisiter les outils de modélisation afin de les adapter à un usage lors de l'exécution. Cette section se conclut ensuite par la description des projets liés en section 10.2 qui sont venus se greffer à Kevoree et qui valident son emploi dans différents cas d'usage, dont une implantation en collaboration avec les sapeurs-pompiers du cas d'usage de motivation décrit en début de cette thèse.

10.1 Concrétisation du modèle et éléments validant l'utilisabilité de Kevoree

Cette section vise a répondre à la question 3 des axes d'évaluation, à savoir donner des éléments quant à l'utilisabilité du modèle mais également des outils proposés pour les concepteurs de systèmes DDAS. La concrétisation des plates-formes Kevoree et des différents langages permettant de modéliser les systèmes seront donc détaillés avant de conclure sur une évaluation expérimentale face à un panel d'ingénieur.

10.1.1 Implantation sous la forme d'un projet *open source*

La qualité d'une abstraction se mesure à sa capacité à résoudre un problème donné [Kru92], d'après Krueger *et al.* cette capacité se concrétise par la distance cognitive entre le pro-

blème et l'abstraction. Les travaux de cette thèse s'inscrivent dans une démarche de recherche d'abstraction pour le génie logiciel, en cherchant à construire un modèle pour les systèmes distribués adaptatifs. L'usage de tels systèmes DDAS est multiple, et de ce fait la validation de l'abstraction permettant de les construire également. Le domaine de simplification d'une abstraction n'est pas facile à trouver, comme le rappelle Guerraoui [GF99]. En effet le plus grand danger pour une abstraction est de cacher les problèmes et de proposer des métaphores non cohérentes avec les problèmes du système. En substance, Guerraoui explique que la plus dangereuse métaphore pour les systèmes à objets distribués est le mythe de la distribution transparente comme cela a pu être le cas avec les RPC, cachant les problèmes et interdisant au développeur de réaliser du code adapté à la distribution métier. Ramené aux DDAS le risque est donc de cacher un aspect de l'adaptation distribuée qui empêcherait l'implantation d'un cas d'usage. Même si l'approche Model@Runtime recherche l'automatisation des systèmes d'adaptation, la couche modèle sert avant tout au développement du système, et est donc largement proche et utilisée par des développeurs humains. L'évaluation du ressenti et la compréhension du modèle d'abstraction proposé sont donc complexes car largement liées à des facteurs humains et dont les critères sont relatifs au cas d'étude et au type de développeur.

A défaut de pouvoir démontrer l'aspect généraliste de la solution, l'évaluation de la solution proposée passe par son usage dans différents projets qui sont venus se greffer à Kevoree . Dans un premier projet, Kevoree a servi à développer une plate-forme de domotique distribuée, dans le cadre d'une collaboration avec Grégory Nain. Le cas d'usage sapeur-pompier détaillé en début de cette thèse a été concrétisé dans un projet nommé DAUM et détaillé au cours de cette section. Enfin la collaboration la plus importante aura été celle qui a permis l'élaboration d'une solution pour la modélisation et l'adaptation de système de type cloud computing dans le cadre d'une collaboration avec Erwan Daubert, doctorant des équipes Myriads et Triskell.

Pour construire ces collaborations et donc avoir un retour sur le ressenti des développeurs vis-à-vis de l'abstraction proposée, il était nécessaire de construire un prototype qui implante le principe du Model@Runtime mais surtout les outils qui permettent son adoption pour la communauté des développeurs. La suite de cette section détaille les éléments de l'outillage mis en place avant de présenter les projets liés, les informations complémentaires sur cet outillage sont disponibles sur le site du projet [1].

10.1.2 Implantation d'outils et langages dédiés pour la construction de composants et manipulation de modèles d'architectures

L'approche Kevoree a été développée à l'aide des outils de l'IDM. Les modèles structurels proposés ont donné lieu à une implantation d'un méta-modèle suivant le standard EMOF [AK03b], [SBMP08]. Les informations liées aux *TypeDefinition* des modèles instances de ces méta-modèles rentrent dans le cadre de la conception continue et font donc l'objet d'un outillage permettant de faire le lien entre le couche de modélisation et la couche de développement. Ces outils sont détaillés en sous-section 10.1.2.3. Les

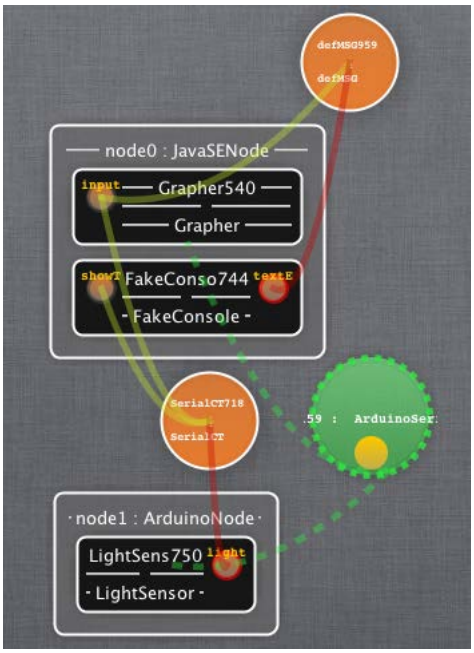
1. http ://kevoree.org/

informations liées aux *Instance* nécessitent des DSL dédiés pour la manipulation et la représentation du modèle d'architecture, ceux-ci sont détaillés en sous-sections 10.1.2.1 et 10.1.2.2.

10.1.2.1 Notation graphique d'architecture

Les notations graphiques sont particulièrement adaptées pour les représentations structurelles car elles permettent de naviguer au milieu des éléments en suivant les relations entre eux, selon les liens de contenance et de liaisons (par exemple lien entre composants). Le modèle Kevoree a donc été enrichi avec un DSL graphique qui permet de représenter et manipuler les notions du modèle à composants. Cette notation graphique est illustrée sur un exemple en figure 10.1.

FIGURE 10.1 – Concepts graphiques du modèle Kevoree



Les composants sont représentés par un rectangle noir avec le nom de l'instance et le nom du type de composant. Les ports requis sont représentés par un rond à droite du composant tandis que les ports fournis sont représentés par un rond à gauche. Les ports de type message ont un fond orange tandis que les ports de type service ont un fond de couleur grise.

Les Channels sont représentés par un rond orange et leurs relations avec les ports sont représentées par un trait, jaune vers un port fourni, rouge vers un port requis.

Les Groups sont illustrés par un rond vert et la relation avec les nœuds (abonnements) est représentée par un trait en pointillés.

Les nœuds sont représentés par un carré gris et les instances qu'il contient à l'intérieur, le titre de ce carré ayant la syntaxe *titrenœud : titreTypeDefinition*.

10.1.2.2 KevScript : DSL de manipulation de modèle d'architecture

Pour gérer le passage à l'échelle, la notation graphique doit s'équiper de mécanismes de *slicing* pour limiter la visualisation à un sous-ensemble du modèle. Pour pallier ce manque et automatiser la manipulation de modèle Kevoree, un DSL textuel a été ajouté au projet. A l'inverse de travaux analogues tels FScript de Fractal, KevScript ne manipule que des modèles et n'a aucun lien avec la plate-forme. Le caractère transactionnel de son exécution est alors uniquement lié au processus d'application du modèle qui résulte de la transformation.

Le langage KevScript est embarqué dans les plates-formes, son but est alors de servir de base pour les raisonneurs qui sont caractérisés par de nombreuses opérations de manipulations de modèles. Pour faciliter ces manipulations, le langage KevScript définit des primitives de haut niveau comme par exemple la migration d'un composant d'un nœud à un autre sous la forme : moveComponent a => nodeB. Le listing 10.1 illustre quelques primitives intéressantes du langage KevScript pour illustrer sa syntaxe. Essentiellement le script se divise en plusieurs familles de méthodes qui sont exécutées de façon séquentielle. D'un côté une famille assure la manipulation des instances du modèle à la manière d'un modèle CRUD (create, remove, update, delete) : addNode, removeChannel. D'un autre côté, une autre famille permet la manipulation des types et ainsi initier une génération de code, comme par exemple l'instruction : createComponentType newFooType.

10.1.2.3 Model2Code et Code2Model

Definition cyclique et lien avec le modèle de dévelopement : exemple avec le langage Java Le lien avec le modèle de développement est primordial pour permettre une utilisation raisonnable de la conception continue. En d'autres termes il doit être possible d'extraire les informations du modèle de développement pour construire le modèle de conception. A l'inverse, il doit être également possible de générer le code à partir du modèle de conception. En se complétant, ces deux processus permettent d'exploiter le modèle à la fois pour la définition initiale d'un *ComponentType* par exemple mais également pour sa compréhension et sa modification après un premier développement pour faire évoluer ses capacités en rajoutant par exemple un port. On trouve de nombreuses solutions dans les travaux liés étudiés dans l'état de l'art de cette thèse. Pour n'en citer que quelques uns, le projet ArchJava propose d'étendre le langage de développement pour y inclure les primitives de conception des composants, tandis qu'une approche telle que Fraclet propose de décorer de façon non invasive le code Java

Listing 10.1– Extrait des commandes KevScript

```
//inclure une definition de modele, depuis un artefact distant ou local
merge "mvn:groupID/artefactID/version"
merge "file:foo.kev"

//manipulation d'instance kevoree
addComponent fooID@node0:FooComponentType
addChannel fooChannel : FooChannelType
addGroup fooGroup : FooGroupType
addNode fooNode : FooNodeType
removeComponent fooID@node0
removeChannel fooChannel
removeGroup fooGroup
removeNode fooNode

//manipulation de dictionaire d'instance
updateDictionary fooID[@node0] {port="8080"}

//manipulation de liaison composant et channel
bind fooComponent.port1@node0 → fooChannel
unbind fooComponent.port1@node0 → fooChannel

//migration de composant fooComponent d'un noeud node1 vers node2
moveComponent fooComponent@node1 → node2

//abonnement de noeuds aux groupes
addToGroup node0@fooGroup
removeFromGroup node0@fooGroup

//manipulation de modele de topologie
network node1 → node2 {"ip" = "192.168.0.1"}

//manipulation et creation de definition de type
createComponentType newFooType
addInPortType newPortName : Message →  newFooType
createChannelType newFooChannel
```

et de le lire à l'exécution. D'autres approches comme les DSL internes de Scala [2] ou Kotlin [3] permettent également d'extraire des informations qui sont ajoutées au langage de développement sans pour autant le modifier directement comme avec ArchJava.

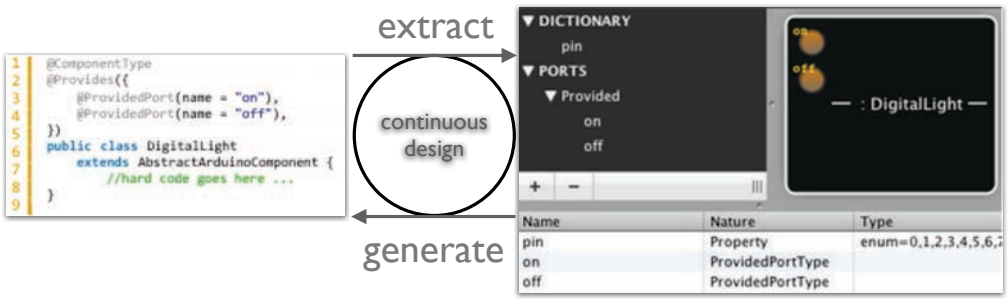
Ce type de méta-information que l'on peut rajouter, tel que les annotations Fraclet, peut être pris en compte au moment de la compilation ou au moment du chargement dans la plate-forme. Dans l'outillage Kevoree le choix s'est porté sur une technique d'annotation non invasive du code, prise en compte directement au moment de la compilation. De ce choix est donc issu un *framework* d'annotations exploitables sur différents langages compatibles sur la machine virtuelle Java , tel que Java, Scala et Kotlin. De plus le fait de prendre les annotations en compte au moment de la compilation permet d'effectuer les optimisations sur le poste du développeur et ainsi permettre le déploiement de Kevoree sur des plates-formes plus modestes telles que Android. Le reste de la section suivante détaille ce *framework*, qui permet de décrire les définitions de type de

2. http ://www.scala-lang.org/
3. http ://kotlin.jetbrains.org/

Kevoree sur du code source Java.

Le but de l'outillage de Kevoree est de proposer un cycle de développement continu. D'un côté l'analyse des méta-données telles que des annotations permet d'extraire le modèle d'un code source et de l'autre il est possible de générer ce même code à partir d'un modèle. La boucle illustrée par la figure 10.2 montre l'usage cyclique des ces deux outils. Le générateur de code n'est pas détaillé ici mais permet de générer l'inverse exact du résultat de l'analyseur de code, c'est à dire un squelette de code source Java symétrique à la définition de type du modèle. Cette génération n'est pas destructive, en d'autres termes elle garde l'ensemble du code ajouté par un développeur précédent et met uniquement à jour les éléments manquants vis-à-vis du modèle.

FIGURE 10.2 – Boucle Modèle vers Code et Code vers Modèle



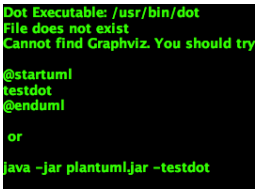
Annotation commune de définition de type L'approche proposée dans Kevoree annotation est fondée sur l'utilisation d'une classe comme point d'entrée pour la définition de type, que ce soit pour un *ComponentType* , un *ChannelType* , un *GroupType* ou un *NodeType* . Il est donc possible de décorer une classe à l'aide d'une annotation pour spécifier le type du point d'entrée, comme illustré dans le listing 10.2.

Listing 10.2– Annotation de déclaration de *TypeDefintion*

```
@ComponentType
public class FooKevoreeComponentType extends AbstractComponentType {}
@ChannelTypeFragment
public class FooKevoreeChannelType extends AbstractChannelType {}
@GroupType
public class FooKevoreeGroupType extends AbstractGroupType {}
@NodeType
public class FooKevoreeNodeType extends AbstractNodeType {}
```

En supplément il est possible d'hériter d'une classe abstraite ou d'une interface pour pouvoir accéder aux méthodes du *framework* Kevoree . Ainsi il est possible pour un composant d'accéder à ses ports, d'utiliser les primitives de renvoi de messages ou encore d'accéder à la couche Model@Runtime embarquée. La figure 10.3 détaille ces classes abstraites par un diagramme de classes.

FIGURE 10.3 – Diagramme de classe de l'API Kevoree pour les *TypeDefintion*



Tous les *TypeDefintion* définissent un cycle de vie, à savoir un état arrêté, démarré, mis à jour (cas de mise à jour de dictionnaire ou de liaisons). Pour la réalisation de ces étapes en Java, il est possible d'annoter une méthode pour chaque étape comme l'illustre le listing 10.3.

Listing 10.3– Annotation de cycle de vie

```
public class FooKevoreeType {
        @Start
        public void start(){}
        @Stop
        public void stop(){}
        @Update
        public void update(){}
}
```

La définition du *DictionaireType* commune également à tous les *TypeDefinition* suit également le même principe. Une annotation *@DictionaryType* est ajoutée pour décorer la classe Java. Cette annotation contient un ensemble de *@DictionaryAttribute* qui définissent les attributs du dictionnaire et leurs attributs (énumération, optionnel, etc). Il est à noter ici l'attribut *fragmentDependant* qui signifie que l'attribut prend une valeur différente suivant le nœud d'hébergement, ceci est illustré sur le listing 10.5.

Listing 10.4– Annotation de dictionaire type

```
@DictionaryType({
    @DictionaryAttribute({name="attID",optional=true}),
    @DictionaryAttribute({name="attID2",vals={"val1","val2"},default="attID2"}),
    @DictionaryAttribute({name="attID3",fragmentDependant=true})
})
public class FooKevoreeType {  }
```

L'héritage de *TypeDefinition* suivant le modèle Kevoree exploite l'héritage du langage à objets hôte, ici Java. Le fait d'hériter d'une classe ou d'une interface Java définissant déjà une annotation de définition de type suffit pour déclarer la relation d'héritage. Bien évidement l'héritage simple de Java est alors une limite pour le modèle Kevoree , et plusieurs solutions sont disponibles pour déclarer un héritage multiple de composants, par exemple par l'ajout d'autres annotations ou en exploitant la notion de trait des langages tel que Scala ou Kotlin mais ceci n'est pas détaillé ici.

Annotation de définition de *ComponentType* et *Port* La description des contrats de composant s'accompagne en plus de la définition de type d'une définition des ports requis et fournis. Cette définition se fait par annotation de la classe, séparée en deux sections : les ports requis et fournis, comme illustré par le listing 10.5.

Les ports requis nécessaires au fonctionnement du composant sont décrits dans une annotation *@Requires*, *via* des champs *@RequiredPort*. Tous les ports peuvent être de type *Service* ou *Message* , optionnels ou non pour le fonctionnement du composant. Dans le cas d'un port *Service* sa description d'interface peut être définie à l'aide d'une interface Java *via* le champ *className*. Il est à noter ici le champ *needCheckDependency* qui définit si un port doit être utilisé ou non dans les phases de démarrage ou d'arrêt, ce qui a des implications pour la planification du déploiement de ce composant.

Listing 10.5– Annotation de dictionaire type

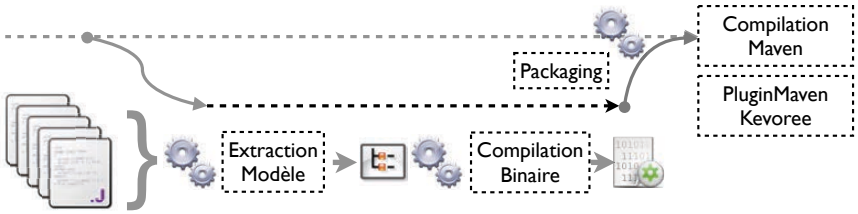
```
@Requires({
        @RequiredPort(name = "out1", type = PortType.MESSAGE, optional = true),
        @RequiredPort(name = "toggle", type = PortType.SERVICE,
                classname = ToggleLightService.class, optional = true,
                needCheckDependency = true)
})
@Provides({
    @ProvidedPort(name = "in1", type = PortType.MESSAGE),
    @ProvidedPort(name = "toggle", type = PortType.SERVICE,
                classname = ToggleLightService.class)
})
@NonConcurrency("in1","toogle")
@ComponentType
public class FooComponentType extends AbstractComponentType {
        @Port{name="in1"}
         public void reactOnMessage(Object o){}
        @Port{name="toogle",method="toogle"}
         public boolean toogleImpl(){}
}
```

La définition des ports fournis suit exactement le même cheminement, le contrat est alors défini dans une annotation *@Provided* et dans des champs *@ProvidedPort*. L'implantation des méthodes des ports est plus complexe. En effet comme nous l'avons publié précédemment [NFM+10], le modèle Kevoree doit intégrer les modèles à services et les communications par évènements. De plus dans le modèle Kevoree rien n'empêche d'avoir plusieurs ports exploitant la même interface message ou service. Ainsi il doit être possible de définir plusieurs ports de service exploitant la même interface Java. Or la plupart des projets liés étudiés dans l'état de l'art exploitent les interfaces Java pour définir les notions de ports de service d'un composant, faisant du même coup l'hypothèse que le composant ne peut définir ces ports plus d'une fois. Pour contourner cette limitation de Java le principe des annotations Kevoree est de séparer la définition des ports fournis en deux parties : d'un côté la définition du contrat, et de l'autre la définition des équivalences entre les méthodes de services et les méthodes Java les implantant. Pour définir ce *mapping* le *framework* définit une annotation *@Port* que l'on peut exploiter pour décorer n'importe quelle méthode Java de la classe du composant

type. Ainsi pour le *mapping* d'un port de type message tel que *in1* dans l'exemple du listing une annotation sans paramètre au-dessus d'une méthode prenant un attribut pour récupérer l'objet en transit suffit. Dans le cas d'un port de type service , il faut alors faire correspondre l'ensemble des méthodes du service fourni vers des méthodes Java comme l'illustre le port *toogle*. Kevoree fait ensuite la redirection des appels de service vers les méthodes appropriées, permettant ainsi de définir autant de ports que nécessaire.

Intégration dans les environnements de compilation Le traitement du code source et des annotations des *TypeDefinition* est réalisé au moment de la compilation pour l'intégration avec la langage Java. Ce traitement consiste principalement à produire un fragment de modèle d'architecture ainsi qu'à ajouter des binaires au *packaging* produit par la compilation (JAR en Java). L'intégration avec les outils de compilation largement utilisés par les développeurs de ce langage est donc indispensable pour rendre le développement des *TypeDefinition* viables et permettre la réutilisation de tous les processus industriels d'intégration continue tels que Jenkins [4]. Le projet Apache Maven [5] définit justement un processus de compilation extensible et est largement utilisé pour la plupart des projets Java depuis plus d'une dizaine d'années. Les phases de compilation Kevoree sont donc intégrées dans le processus Maven sous la forme d'un *plugin* dont une illustration du processus est donnée par la figure 10.4.

FIGURE 10.4 – Processus de compilation Kevoree par extension de Maven



De part cette intégration les fichiers Java annotés sont traités directement pendant la phase de packaging et permet ainsi de manière transparente pour le développeur de rendre compatibles les JAR produits par ses projets pour Kevoree . L'intégration avec l'environnement Maven de Kevoree va bien plus loin que ce simple processus puisqu'elle propose également une lecture des fichiers projets Maven mais également une synchronisation du code utilisateur avec un modèle,etc... La description technique de ces fonctionnements n'est pas détaillée ici mais est néanmoins nécessaire pour l'adoption de l'abstraction par les développeurs Java.

10.1.2.4 Kevoree IDE, environnement de modélisation d'architecture

L'ensemble des outils de modélisation Kevoree ont été intégrés dans un environnement de développement dédié à la manipulation de modèles Kevoree . Cet environne-

4. http ://jenkins-ci.org/
5. http ://maven.apache.org/

ment illustré par la figure 10.5 peut être exploité seul mais également dans des environnements de développement généraliste tels que Eclipse, Netbeans ou encore IntelliJ. En reprenant le DSL graphique et textuel, il peut lire et modifier à distance les modèles embarqués dans des nœuds de différentes implantations.

FIGURE 10.5 – Environement de modélisation d'architecture Kevoree

Cet environnent a deux buts : d'un côté le DSL graphique a un aspect pédagogique fort car il permet de réduire la courbe d'apprentissage du modèle à composants, et d'un autre côté il se place en soutien pour la conception des composants lorsqu'il est rapproché d'un environnement de développement généraliste.

10.1.3 Passage de l'abstraction à l'implantation : adaptation des outils de modélisation pour une exploitation à l'exécution

L'objectif final de l'approche Kevoree est le pilotage d'un système adaptatif par une couche de modélisation. Mais cette couche de modélisation doit être disponible et exploitable par les éléments déployés sur le système pour rendre la couche de réflexion concrète. Cependant le passage du modèle à la plate-forme ne se fait pas sans des ajustements, car les outils de modélisation n'ont pas été développés pour être déployés directement dans un système. Dans le cadre de cette thèse, ce constat a entraîné des travaux pour adapter ces outils et rendre viable le déploiement d'un modèle dans un système. Le but premier du système est la fonctionnalité métier à réaliser , qui ne doit être que peu influencée par l'adaptation dynamique dirigée par le modèle. La technologie de modélisation embarquée doit de ce fait limiter son impact et le *framework Kevoree Modeling Framework* a été développé dans ce sens. Il est détaillé à la section 10.1.3.1. Le reste de cette sous-section détaille les implantations des différentes plates-formes qui

permettent de concrétiser les nœuds d'exécutions sur différents environnements.

10.1.3.1 Kevoree Modeling Framework

Le framework KMF est né d'une volonté de fournir une technologie de modélisation permettant de rendre viable l'usage du Model@Runtime sur des systèmes même modestes en terme de capacité de calcul, tels que des *smart phones*. Ces travaux ont fait l'objet d'une publication à la conférence MODELS'2012 [FNM+12]. En effet pour exploiter les modèles directement par les éléments de la plate-forme il est nécessaire de réduire le surcoût de ceux-ci et notamment en terme d'occupation mémoire et surtout en temps de parcours par les différents raisonneurs.

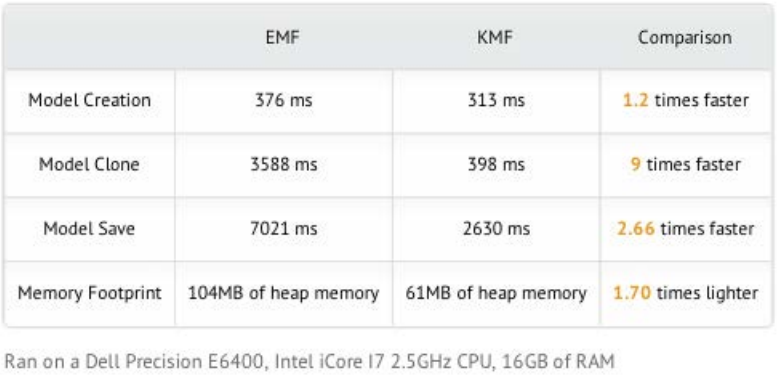
En plaçant le modèle au centre de la plate-forme dirigée par les modèles, celui-ci devient une ressource partagée par les différents composants et ce modèle doit alors faire face à trois défis :

- **Les accès concurrents :** le modèle d'architecture courant est lu par plusieurs raisonneurs en même temps et doit donc être capable de se protéger contre les lectures concurrentes.
- **La capacité à se dupliquer :** le Model@Runtime est caractérisé par des opérations *offline* et *online* et doit donc permettre de désynchroniser le modèle de la plate-forme. Pour l'implantation de cette désynchronisation des opérations permettant de cloner le modèle sont nécessaires et donc l'efficacité doit permettre un usage pour chaque tentative de modification.
- **La capacité de sérialisation/désérialisation :** dans un usage distribué le modèle doit être échangé sur les différents nœuds, et donc avoir un opérateur de (dé)sérialisation qui doit donc être à la fois rapide et peu consommateur de ressources pour être exploitable sur des nœuds de puissance modeste.

Le standard de fait exploité par la majorité des outils de modélisation suit le MOF ; l'implantation la plus connue est EMOF, par la communauté Eclipse. Cependant l'implantation d'EMOF ne permet pas des performances raisonnables pour les besoins exprimés ci-dessous, introduisant du même coup un surcoût pour l'introduction de cette technologie au centre d'une plate-forme à composants. Dans la publication [FNM+12], nous avons proposé une implantation respectant ces principes et permettant d'avoir une couche de modélisation efficace pour la plate-forme Java, et exploitant des technologies récentes telles que Scala. Un extrait de ces résultats est illustré par la figure 10.6 : il s'agit des améliorations sur les opérateurs correspondant aux besoins exprimés ci-dessus dans le cas d'un modèle d'automate à état de 100 000 états.

Ces travaux explorent également les besoins en chargement paresseux nécessaire dans ce cas d'usage. KMF a été développé dans le cadre de cette thèse pour pouvoir répondre aux besoins des plates-formes embarquées telles que les *smart phones* Android ou les Raspberry PI. Ces travaux se poursuivent désormais pour traiter des modèles à très large échelle et ainsi anticiper l'utilisation du Model@Runtime sur les systèmes très large échelle.

FIGURE 10.6 – Résultat expérimentaux de KMF sur le cas d'usage FSM



	EMF	KMF	Comparison
Model Creation	376 ms	313 ms	1.2 times faster
Model Clone	3588 ms	398 ms	9 times faster
Model Save	7021 ms	2630 ms	2.66 times faster
Memory Footprint	104MB of heap memory	61MB of heap memory	1.70 times lighter

Ran on a Dell Precision E6400, Intel iCore I7 2.5GHz CPU, 16GB of RAM

10.1.3.2 Implantation des nœuds

Le principe même de la conception continue exige que le système soit capable de faire du déploiement à chaud des artefacts logiciels. L'implantation des nœuds type doit donc non seulement fournir l'environnement permettant de faire la comparaison mais aussi de déployer du code. Cette sous-section traite de façon non exhaustive les différents environnements dirigés par Kevoree développés dans cette thèse.

NodeType pour les plates-formes Java, OSGi L'implantation de Kevoree étant réalisée en Java et Scala, la plate-forme JavaSE constitue le nœud par défaut dans l'approche. Les artefacts de développement et de déploiement sont alors des fichiers de type JAR contenant des classes compilées prêtes à être intégrées dans le système.

Dans une première version la plate-forme Kevoree a été implantée au-dessus du modèle OSGi afin de pouvoir s'intégrer et piloter des environnements existants déjà construits sur cette plate-forme. Le nœud type Kevoree OSGi ainsi développé permet de prendre comme unité de déploiement les *bundles* OSGi construits par les développeurs auxquels s'ajoutent les méta-informations.

Cependant le non-alignement du modèle Kevoree et le modèle OSGi en terme de modèle de développement a posé de nombreux problèmes de compréhension aux utilisateurs de ces composants. En effet le modèle de développement et les modèles de projet tels que Maven, SBT, Ivy, etc se fondent sur la notion de dépendance qui s'exprime en lien entre JAR. Cette notion de dépendance n'est pas alignée avec la notion de *manifest* OSGi qui exploite une relation entre les packages Java pour exprimer les dépendances.

Le projet Kevoree ClassLoader (KCL) a été introduit pour simplifier le développement des composants Kevoree et pour permettre une plus grande flexibilité. Ce sous-projet permet de réaliser le chargement à chaud de JAR ainsi que leurs liaisons dynamiques, en prenant un graphe de dépendances. Le nœud standard Kevoree JavaSE exploite donc cette technologie sous-jacente en remplacement de *frameworks* OSGi et permet ainsi un alignement total entre le modèle de développement Java et Kevoree .

Le chargement à chaud de code pose de nombreuses questions de sécurité qui sont

abordées dans les perspectives de cette thèse.

NodeType Android, plateforme Dalvik La plate-forme Android exploite une machine virtuelle nommée Dalvik [Ehr10]. Cette dernière exploite un *bytecode* similaire à celui de Java, à ceci près qu'elle exploite une machine à registres là où la JVM exploite une machine à pile. La conversion du *bytecode* Java vers Dalvik est donc possible, et pour l'implantation du nœud Android ceci nous a permis de reprendre les résultats de la plate-forme JavaSE pour la comparaison de modèle. Le chargement à chaud du nouveau *bytecode* est cependant différent : celui-ci a été réalisé comme extension du projet KCL afin que ce dernier puisse charger les fichiers de base Android (.dex) et les lier entre eux suivant un graphe de dépendance. Ainsi les modèles de développement de Java et Android sont alignés grâce au chargement transparent de KCL sur les deux environnements.

NodeType Arduino Le nœud Arduino est lui développé en C et en langage Processing[6]. Son développement est séparé en deux car son processeur ne permet pas de prendre en compte les comparaisons de modèles. Celles-ci sont donc effectuées sur un nœud parent capable de faire tourner l'algorithme de comparaison Java. Le nœud Arduino ne permet pas non plus de faire du chargement à chaud de code, cependant il contient un *framework* d'instanciation dynamique. Pour cela un *framework* spécifique ainsi qu'un ordonnanceur de composant ont été développés afin de permettre l'instanciation de composants et leur planification à chaud.

10.1.3.3 Maturité du projet

Le projet Kevoree est un projet *open source* dont le développement a commencé début 2010 et est toujours en développement actif. Le module KevoreeCore définit l'ensemble des éléments de modélisation ainsi que les opérateurs de comparaison et fusion de modèles d'architecture. Au moment d'écrire ces lignes, ces modules contiennent 42210 lignes de code (Java,Scala,XML confondu) sans compter les parties générées par le *framework* KMF. Le module KevoreeTools définit lui l'ensemble des outils destinés à la conception continue sur les différentes plates-formes. Ce module contient 67090 lignes de code (Scala, XML, Java, C, C++ confondues). Le module KevoreePlatform définit les différentes plates-formes dynamiques capables de déployer le code des composants (JavaSE, Android, Arduino, etc). Ce module contient 26908 lignes de code (XML, Java, Scala, shell).

Enfin le projet Kevoree fournit deux bibliothèques de composants disponibles sur étagère, contenant l'ensemble des composants définis dans les différents cas d'usage. La première est la bibliothèque standard, contenant des nœuds et *channels* types de base nécessaires pour toutes les orchestrations. Cela représente 115016 lignes de code (hors Html et javascript), et représente 76 *ComponentType* , 13 *ChannelType* , 11 *NodeType* et 7 *GroupType* . La deuxième bibliothèque contient des composants plus expérimentaux.

6. http ://processing.org/

Cela représente environ 100000 lignes de code, et représente 37 *ComponentType* , 8 *ChannelType* , 7 *NodeType* et 5 *GroupType* .

La figure 1 donnée en annexe de cette thèse représente le diagramme de dépendance de Kevoree.

10.1.4 Évaluation de prise en main par un panel de chercheurs et d'étudiants

L'évaluation d'une abstraction est particulièrement difficile et notamment sa perception par les utilisateurs. Qui plus est dans le cas d'une abstraction pour le développement, la qualité de l'abstraction devrait se mesurer *via* le taux de fautes évitées ou encore le gain en temps de développement. A défaut de pouvoir faire ce type d'évaluation empirique dont le terme dépasserait le temps de cette thèse, le choix s'est porté sur une validation par les cas d'usage aux limites précédemment exposées. Cependant afin d'améliorer et d'évaluer de manière empirique la qualité des outils fournis, plusieurs travaux pratiques ont été réalisés, avec des étudiants tout d'abord puis avec un panel de chercheurs pour avoir un retour sur la qualité de l'abstraction.

Au moment d'écrire ces lignes, trois travaux pratiques de 4 à 8 heures ont été réalisés avec des étudiants de dernière année en Master d'informatique (ISTIC et ESIR, université de Rennes 1). Dans les deux cas le but était de faire concevoir un cas d'usage par des développeurs de niveau ingénieur, en mettant en oeuvre plusieurs nœuds de calcul et des adaptations dynamiques .

Les résultats de ces premières expériences ont permis de démontrer que la courbe d'apprentissage avec l'abstraction proposée est bonne puisque tous les groupes de TP ont réussi à réaliser la coordination de plusieurs nœuds en moins de 2 h de travail. De même, les premières adaptations mettant en oeuvre des reconfigurations structurelles simples (déplacement d'un composant) requièrent moins de 2 h d'apprentissage.

De façon plus chiffrée, une autre expérience de tutoriel a été réalisée sur un panel de 20 chercheurs dans le cadre d'un séminaire d'un réseau d'excellence. Le tutoriel était alors plus ambitieux puisqu'il proposait des développements qui allaient de l'assemblage de composants à la réalisation d'un serveur élastique déployé sur une plate-forme MiniCloud (simulation d'une infrastructure Cloud en machine virtuelle Java)

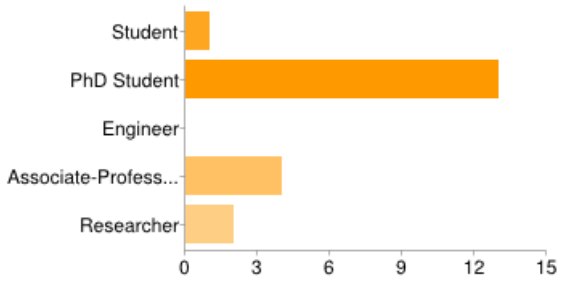
La figure 10.7 représente la répartition des différents statuts des participants qui sont majoritairement des étudiants en thèse.

Les compétences du panel de participants étaient très variables en terme de développement Java et en connaissance des approches à composants, comme l'illustrent les figures 10.8 et 10.9.

Les compétences sur le système de gestion de projet Maven qui a été utilisé dans le tutoriel étaient majoritairement à acquérir par le panel, comme l'illustre la figure 10.10. De même, la plupart des participants n'avaient que peu d'expérience de Kevoree , comme l'illustre la figure 10.11.

Après 2 h de cours puis 2 h de travaux pratiques, la plupart des participants ont réussi à développer et à assembler des composants de manière distribuée. La dernière tâche d'élasticité sur une infrastructure de type *cloud* nécessite cependant plus de temps

FIGURE 10.7 – Statuts des participants à l'expérience empirique



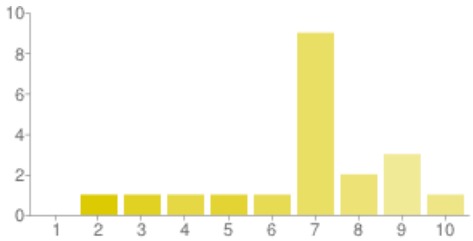


FIGURE 10.8 – Compétence en Java des participants (0 la plus faible)

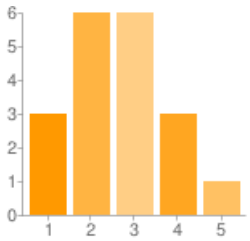


FIGURE 10.9 – Compétence en CBSE des participants (0 la plus faible)

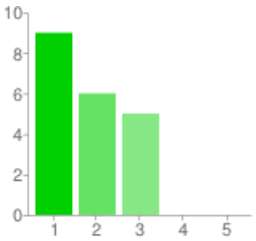


FIGURE 10.10 – Compétence en MAVEN des participants (0 la plus faible)

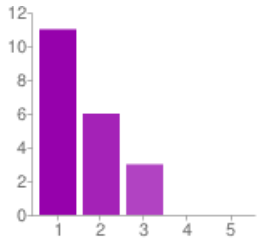


FIGURE 10.11 – Compétence en Kevoree des participants (0 la plus faible)

puisqu'une seule personne a atteint ce but dans le temps donné comme l'illustre la figure 10.12. Le langage graphique de modélisation d'architecture a été largement compris par la majorité des participants et semble donc adapté à la représentation du problème comme l'illustre la figure 10.13.

Globalement les participants ont indiqué que la complexité de création d'un nouveau *ComponentType* est légèrement supérieure à la création d'une classe Java et est donc abordable (Figure 10.14). L'utilisation du projet KCL pour rapprocher le modèle de déploiement du graphe de dépendance Maven a été également en majorité bien compris par les participants (Figure 10.15).

Enfin, la figure 10.16 illustre les réponses des participants à la question : quelle est la complexité de création d'un groupe type, par exemple en utilisant un algorithme de type Paxos. Cette fonctionnalité de Kevoree est de loin la plus difficile, à cause

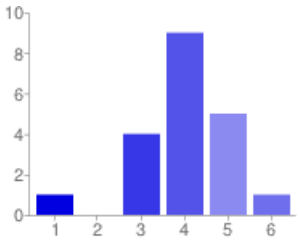


FIGURE 10.12 – Réussite de l'ensemble du tutoriel (6 réussite totale)

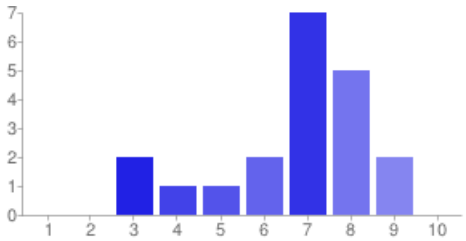


FIGURE 10.13 – Compétence en Kevoree des participants (10 compréhension parfaite)

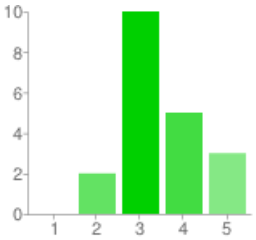


FIGURE 10.14 – Facilité de création de composant (5 comme une classe Java)

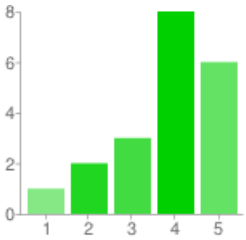
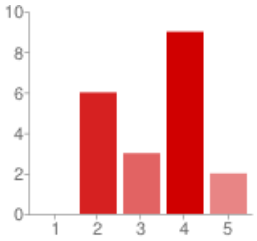


FIGURE 10.15 – Compréhension du rôle de Maven et du graphe de dépendance sur le déploiement (5 parfaitement compris)

de la complexité algorithmique de la distribution tout d'abord, mais également pour la gestion de la fragmentation telle qu'elle est définie dans cette thèse. Cependant, à l'issue du tutoriel, les participants répondent à 45% 4/5 à cette question, indiquant qu'ils sont proches de faire ce type d'implantation et 10% ont déjà commencé à modifier le code des groupes.

FIGURE 10.16 – Complexité de création d'un GroupType (5 étant prêts pour l'écriture d'un Paxos)



Ces deux tutoriels donnent un premier niveau de confiance quant à la maturité de l'outil proposé mais surtout quant à la complexité de prise en main de telles abstractions. En effet la plupart des participants n'avaient qu'une connaissance limitée dans la conception d'architecture à base de composants. Malgré la prise en main des outils annexes tels que Maven, la courbe d'apprentissage courte a permis à la plupart des participants d'arriver à la construction d'un système distribué ouvert et dynamique.

Un résultat intéressant est le fait que bien que Kevoree ne cache pas ses communications sous un bus abstrait mais au contraire expose les *ChannelType* et les *GroupType* dans son modèle, les participants ont malgré tout réussi à distribuer leurs composants tout en ayant une bonne confiance dans leurs architectures. L'abstraction ne cachant pas la distribution n'effraie donc pas les utilisateurs, qui au contraire ont apprécié le fait de visualiser les interconnexions entre nœuds.

10.2 Validation de la versatilité par la généralisation via des collaborations

Les sous-sections suivantes détaillent les différents projets qui ont exploité et validé le modèle Kevoree dans différents cas d'usage.

10.2.1 Projet Entimid

Le projet Entimid est issu de la thèse de Gregory Nain [NDBJ08a],[NBF+09],[NFM+10],[Nai11]. Entimid est une plate-forme dynamique à composants dédiée à l'intégration de services domotiques pour la réalisation de scénarios complexes tels que l'assistance de personne en dépendance à domicile. Derrière les problèmes métiers du cas d'usage, cette thèse soulève le problème d'interaction entre composants développés avec des interfaces incompatibles, comme c'est souvent le cas lorsqu'on intègre des composants issus de l'IoT et l'IoS. D'un côté les interfaces sont orientées connectique électronique et de l'autre côté elles sont orientées services ; l'intégration des deux manières impose de se reposer la question du typage des interfaces de composants.

D'abord développé sur une plate-forme *ad-hoc* le projet Entimid a ensuite été porté au-dessus de Kevoree en profitant du même coup du modèle mixte service et évènement. Les notions de typage par *duck typing* spécifiques au contexte de la domotique ont pu être implantées et intégrées dans un *framework* étendu de Kevoree .

Le projet Entimid est caractérisé par des technologies très hétérogènes qui doivent être intégrées sans pour autant que l'abstraction proposée masque les couches basses du *middleware*. Deux stagiaires issus de la formation DRI de Rennes (informatique spécialité domotique) ont exploité cette solution lors d'un stage de fin d'étude (Nicolas Richie, Jacky Bourgeois). Ces travaux ont exploré l'usage de l'abstraction proposée sur des cas complexes tels que l'administration et la configuration de périphériques domotiques à distance, ou encore la réalisation d'une centrale de surveillance vidéo au-dessus de Entimid/Kevoree.

En substance, Entimid a permis d'évaluer la dynamicité de Kevoree pour un profil d'utilisateurs de niveau ingénieur sur des cas d'usage hétérogènes mais faiblement distribués. Ces travaux ont largement permis d'améliorer l'outillage pour réduire la courbe d'apprentissage sur ces cas de systèmes dont la mise sur le marché doit être très rapide et implique de développer suivant le même rythme.

10.2.2 Projet DAUM

Après la phase d'étude des besoins, menée courant 2010 sous la direction de Noël Plouzeau et avec les pompiers du SDIS 35, la décision a été prise de concrétiser le projet par un démonstrateur. Le projet DAUM pour Dynamic Adaptation Using Models a donc été initié en ce sens sous la forme d'une action INRIA ADT joignant les efforts des deux équipes de recherche Triskell et Myriads à Rennes.

L'objectif du projet DAUM est donc de réaliser un démonstrateur d'un système tactique de terrain ayant les capacités dynamiques pour faire face aux scénarios des situations d'urgence. Ainsi ce système hautement distribué a pour pré-requis de devoir s'étendre dynamiquement suivant les arrivées de personnels, de matériels (capteurs ou tablettes) ou même en cas de perte de communication réseau. Ces adaptations doivent pouvoir réorganiser le réseau maillé de terrain suivant la hiérarchie métier des personnels sur place.

Constitué à la fois de nœuds mobiles attribués aux personnels et de nœuds de calculs fixes pour la sauvegarde des données, le système DAUM doit permettre d'aider les pompiers à gérer les ressources matérielles (véhicules principalement) et humaines afin de mieux coordonner les situations d'urgence. L'environnement visé est constitué des différents nœuds décrits dans le cas d'usage : des tablettes fonctionnant avec le système Android pour l'interface utilisateur de terrain, et des nœuds de soutien pour le stockage et la répartition de la charge des données. Les personnels sont de plus équipés de nœud comportant des capteurs biométriques. La situation tactique de terrain est donc une agrégation des positions de différents acteurs et matériels ainsi que leurs informations d'état (par exemple état de stress d'un pompier calculé avec son rythme cardiaque et sa chaleur corporelle).

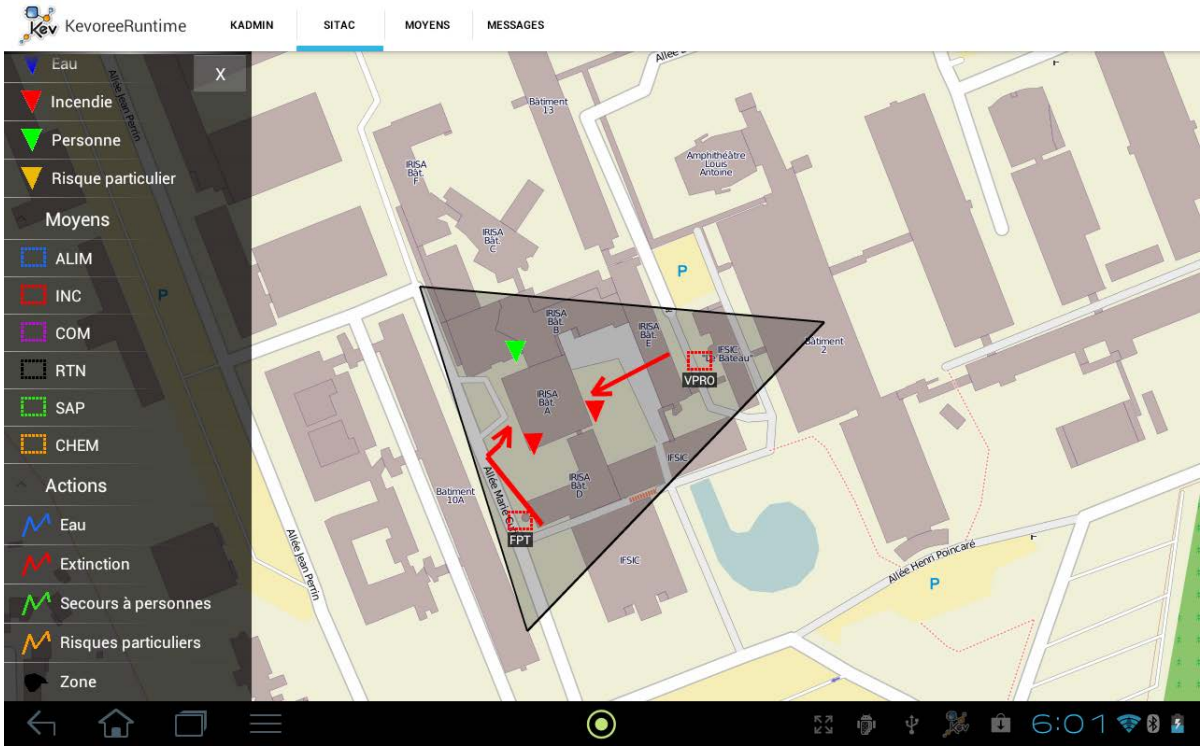
L'approche Model@Runtime proposé par Kevoree a été retenue pour la conception de l'architecture et la gestion de son adaptation durant les interventions d'urgence. Des travaux ont été initiés avec deux stagiaires de niveau Master pour la réalisation de l'interface dédiée à l'édition de situation tactique. Ce sous-projet nommé Agetac a été réalisé sur la plate-forme Android.

Dans le cadre de l'ADT un ingénieur de recherche, Jean-Emile Dartois, est venu en support pour le développement du prototype, accompagné d'un stagiaire de Master, Maxime Tricoire. Dans ce cadre les nœuds type Android de Kevoree ont permis le développement du système de tablette permettant l'édition collaborative de la situation tactique. La figure 10.17 illustre par une capture d'écran l'interface développée.

Le développement de capteurs embarqués dans les vêtements a été réalisé en utilisant des nœuds Kevoree de type Arduino. Ces capteurs embarqués permettent de surveiller les pulsations cardiaques du porteur mais également ses déplacements afin de réaliser dynamiquement ce qui se nomme une balise homme-mort [7]. Ainsi interconnectée avec le système Agetac ces nœuds embarqués peuvent effectuer des adaptations pour changer dynamiquement les capteurs à démarrer, leurs valeurs seuil de déclenchement ou encore leurs algorithmes de suivi . Enfin les nœuds Java standard de Kevoree ont servi au développement des nœuds de soutien embarqués dans les véhicules et stockant les

7. http ://fr.wikipedia.org/wiki/Protection_du_travailleur_isol%C3%A9

FIGURE 10.17 – Interface SITAC propulsé par Kevoree Android



données à l'aide de matériel RaspberryPI [8]. Plusieurs démonstrations ont été présentées aux sapeurs-pompiers du SDIS 35, principalement dans le but de fournir du matériel servant à la formation des futurs agents de terrain avec ces technologies ou même pour leur faire simuler des situations d'urgence en mode *serious game* [JVM05]. Une de ces démonstrations a été publiée et présentée à la conférence UbiMob2012 [FDP+12a]. Au moment d'écrire ces lignes il reste encore un an de développement pour le projet DAUM et la collaboration avec le SDIS 35 est toujours d'actualité.

Le projet DAUM exploite plusieurs aspects du modèle Kevoree . Tout d'abord l'usage de l'abstraction par différents niveaux de compétence a bien permis de mettre en commun les différents composants pour la réalisation d'adaptation sur les différents plates-formes. De plus DAUM exploite des nœuds de natures très différentes, des capteurs aux nœuds Android, et valide donc l'usage du modèle Kevoree pour la construction d'adaptations hétérogènes.

8. http ://www.raspberrypi.org/

10.2.3 Kevoree pour le *cloud*

10.2.3.1 Considérer les *clouds* comme un DDAS hiérarchique

Dans le cadre d'une collaboration avec Erwan Daubert dont la thèse devrait être publiée d'ici fin 2012, le modèle Kevoree a été utilisé pour la gestion de système de type *cloud computing*. L'idée principale de KevoreeCloud est d'utiliser les *NodeType* pour représenter les différents niveaux du *cloud* : IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*), SaaS (*Software as a Service*). La diversité des implantations des *NodeType* permet alors de modéliser les différentes implantations de virtualisation qui sont la base des systèmes *cloud* mais surtout les différentes capacités d'adaptation que l'on peut proposer à différents niveaux.

En d'autres termes un nœud de type IaaS peut héberger et donc ajouter et supprimer des nœuds de type PaaS. Les implantations peuvent exploiter différents fournisseurs d'infrastructure tels qu'Amazon EC2 ou encore Rackspace. Les nœuds PaaS sont eux chargés d'héberger et donc ajouter et supprimer des nœuds de type SaaS mais également des composants servant à fournir un *framework* aux applications du SaaS. Par exemple un composant base de données NoSQL peut être hébergé à ce niveau et partagé entre les composants hébergés au niveau du SaaS. Le niveau SaaS permet lui d'héberger des composants utilisateurs et correspond aux *NodeType* détaillés précédemment dans cette thèse.

Cette notion d'hébergement de nœuds permet de représenter les niveaux de virtualisation du système. En effet chaque nœud est responsable dans ce modèle de ses propres adaptations, et délègue les adaptations qui concernent ses fils. En résumé un nœud est responsable uniquement des instances qu'il héberge et ceci vaut également pour ses nœuds fils dont il ne contrôle que le cycle de vie.

Avec ce principe de délégation et toujours à l'aide de l'approche Model@Runtime il est alors possible d'effectuer des adaptations aux différents niveaux. Par exemple une adaptation déclenchée pour une raison de performance peut jouer à la fois sur l'élasticité de l'IaaS et du SaaS, par exemple en démarrant un nouveau nœud d'hébergement et en y migrant des composants du SaaS. Ce type d'adaptation aujourd'hui complexe à réaliser en raison de la diversité des *clouds* et surtout de l'isolation entre couches est désormais possible.

L'isolation de la visibilité est cependant nécessaire : pour pallier cela des *GroupType* spécifiques ont été développés afin de découper (*slicer*) les modèles qui sont offerts aux nœuds de chaque niveau. En résumé ces groupes permettent de ne synchroniser à un SaaS que la partie utilisateur d'un modèle , en lui cachant la partie du modèle qui représente l'IaaS et le PaaS.

La couche modèle désormais présente dans ce type d'infrastructure peut donc offrir les mêmes services que pour les DDAS mobiles. Ainsi il est possible d'exploiter la capacité d'introspection du *cloud* pour calculer les adaptations afin d'obtenir les propriétés d'élasticité qui le caractérise, par exemple pour répartir les nœuds SaaS sur les différents PaaS et IaaS afin de répartir la charge ou optimiser les temps de réponse des applications.

10.2.3.2 Du *cloud* au *sky computing* dirigé par les modèles

Pour valider cette nouvelle approche, plusieurs *NodeType* ont été développés. Ces nœuds types exploitent l'extensibilité des *NodeType* pour étendre la comparaison de modèle afin de détecter les ajouts, suppressions et mises à jour de nœuds fils.

Un *NodeType* a été implanté tout d'abord au-dessus du fournisseur d'IaaS Amazon, en exploitant l'API EC2. Celui-ci peut alors réaliser ses primitives d'adaptation et créer ou supprimer des machines virtuelles sur cette infrastructure par simple ajout de nœud dans le modèle.

Par la suite, un *NodeType* nommé MiniCloud a également été développé afin de proposer un niveau de virtualisation plus léger et destiné à des infrastructures plus modestes, en réalisant la virtualisation à l'aide de plusieurs machines virtuelles Java et donc en partageant le même système d'exploitation. Enfin ces travaux ont été étendus dans un projet nommé KevoreeKloud, qui construit une infrastructure d'hébergement entièrement dirigée par cette approche et exploitant le système FreeBSD et sa fonctionnalité de *jail* [9] pour la réalisation de la couche de virtualisation. Cette infrastructure de type *cloud* dirigée par les modèles permet d'offrir un lien entre les outils de modélisation et les systèmes *cloud*, notamment pour le calcul des adaptations élastiques. La modélisation fournit un environnement plus maîtrisable, plus léger et moins cher qu'un système géré manuellement, et l'infrastructure KevoreeCloud illustrée en annexe a été développée dans ce sens. Ainsi en assemblant des processeurs basse consommation de type Atom et une paravirtualisation (partage du noyau système, différent de l'hypervirtualisation), le KevoreeCloud limite au maximum le surcoût en garantissant l'isolation décrite dans le modèle.

10.2.3.3 Gérer les *clouds* comme des DDAS, une validation de la distribution large échelle et hétérogène

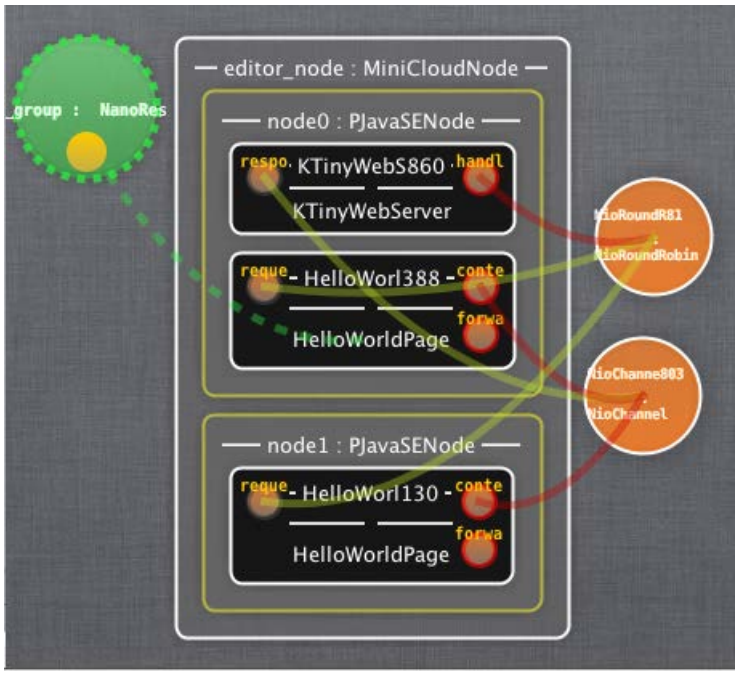
Les architectures de type *cloud computing* sont donc une catégorie de DDAS caractérisée par une forte hétérogénéité des nœuds pour répondre aux besoins divers des plates-formes clients à héberger. De plus ces architectures sont fortement distribuées et surtout nécessitent de nombreuses adaptations pour réaliser l'élasticité nécessaire à l'optimisation de la charge utilisateur. À l'inverse du cas d'usage précédent, ces architectures ne sont pas constituées de nœuds mobiles et ne permettent pas de valider l'adaptation dans un environnement divergent.

La réalisation des algorithmes d'élasticité du *cloud computing* à l'aide du modèle Kevoree et plus généralement à l'aide de l'approche Model@Runtime permet de valider le rapprochement de ces infrastructures avec les architectures de type DDAS. En effet, plus que jamais les décisions de reconfiguration du *cloud computing* nécessitent un modèle de réflexion pour la prise de décision. De plus, l'architecture de type *cloud computing* nécessite des processus de synchronisation de modèle de type consensus, non détaillés ici, mais offre un autre cas d'usage à la notion de groupe.

Enfin, les groupes du *cloud* ont également intégré une notion de *slicing* afin de

9. http ://www.freebsd.org/doc/fr/books/handbook/jails.html

FIGURE 10.18 – DSL graphique étendu pour la représentation des nœuds Cloud



réaliser l'isolation des modèles de réflexion. Cette notion qui touche à la sécurité des approches Model@Runtime mérite largement d'être poursuivie et est abordée dans la section perspective de cette thèse.

En substance le KevoreeCloud a permis de valider le modèle Kevoree dans des expériences large échelle et permet de faire sortir les adaptations du cadre des DDAS conventionnel en introduisant la notion de virtualisation.

Chapitre 11

Conclusion de validation

Le but majeur du projet Kevoree est de fournir un environnement de modélisation, développement et déploiement pour les systèmes distribués adaptatifs hétérogènes. Son évaluation s'est donc déroulée en trois axes. Le premier a permis d'étudier les cas aux limites de cette gestion de noeuds hétérogènes au travers des micro-contrôleurs. Ainsi même dans ces cas de figure embarqués l'adaptation dynamique s'avère compatible avec les besoins en performance des cas d'usage associés en terme de latence et surcoût mémoire notamment. Le deuxième axe s'est lui attardé à montrer que la notion de groupe intégré dans le modèle Kevoree permettait bien d'encapsuler les algorithmes du monde distribué tel que Gossip et que l'ensemble permettait bien d'écrire des systèmes DDAS divergents et convergeants. Enfin le troisième axe s'est attardé à montrer que le développement de tels systèmes DDAS est non seulement réaliste mais également concrétisé dans un projet open-source exploité depuis par d'autres projets dans le cadre de multiples collaborations. Pour initier ces collaborations il a fallu non seulement mettre en place un outillage autour de l'approche Model@Runtime mais surtout adapter les outils de design pour les rendre exploitables directement dans la plate-forme. Finalement le cas d'usage de motivation sapeur-pompiers développé en début de cette thèse donnera lieu à un projet dédié nommé DAUM, détaillé dans la partie perspective de cette thèse. Ainsi le projet DAUM reprend ces résultats tant sur l'aspect micro-contrôleur que sur la distribution de l'adaptation pour réaliser l'implantation du système tactique de terrain ayant les capacités de s'adapter aux besoins de son environnement.

Quatrième partie

Conclusion et perspectives

Vous n'êtes pas fini tant que vous n'avez pas fini de grandir et de changer.
Benjamin Franklin

Cette dernière partie conclut ces travaux de thèse. Il sera tout d'abord rappelé rapidement le contexte de ces travaux de recherche qui abordent les défis induits par le développement et la conception de systèmes tels que les DDAS et CPS. Une conclusion sera alors proposée afin de résumer la contribution et son évaluation par rapport aux axes représentatifs des défis de ce contexte. Enfin cette conclusion se poursuivra sur un paragraphe d'ouverture, qui détaillera les perspectives envisagées à la suite de cette thèse dans des domaines aussi variés que les types de nœuds des CPS et cherchant à rapprocher les CPS des architectures de type *cloud*.

Chapitre 12

Conclusion

12.1 Une convergence des systèmes distribués par nature hétérogènes

Via le système tactique des sapeurs-pompiers motivant cette thèse est illustrée toute une classe de systèmes informatiques qui s'apparentent aux *Cyber Physical Systems*. Les architectures de ces CPS résultent d'une convergence et d'une cohabitation de l'informatique conventionnelle, principalement conçue pour du traitement de données, et de l'informatique embarquée qui permet la connexion à son contexte physique [Lee06a]. Pour répondre à une plasticité grandissante en terme de fonctionnalités, et à des positions de plus en plus critiques, ces systèmes d'information sont considérés et développés comme des systèmes dits *éternels*. Ainsi dès leur conception, ces systèmes intègrent la capacité à se mettre à jour en cours de fonctionnement, en rétro-action à une analyse de leur état ou de leur contexte, comme cela a été décrit dans les premiers processus MAPE d'IBM posant les jalons des systèmes dits *adaptatifs*. Ces systèmes DDAS sont caractérisés par une forte hétérogénéité de leurs nœuds de calcul en terme de types et de puissances, mais surtout par une distribution de leurs traitements, données et capacités d'adaptation.

12.2 L'abstraction en réponse à la complexité de conception

La conception et la maintenance d'un DDAS induit donc inévitablement une complexité due à l'empilement des préoccupations. La montée en abstraction est une réponse communément exploitée que ce soit par la communauté de l'informatique distribuée [LTB08] mais également pour les systèmes adaptatifs [MBJ+09a], [GT04]. L'abstraction permet alors la simplification d'un aspect ou point de vue du problème pour en faciliter sa solution [FKN+92]. Ainsi les concepts de composants et services permettent de représenter respectivement une fonctionnalité métier et des échanges entre processus, par extension des modèles tels que SCA [SMF+09b] qui exploitent ces no-

tions pour construire une abstraction du système dans sa globalité. C'est dans cette optique d'abstraction dédiée aux systèmes adaptatifs qu'a été proposé le paradigme du Model@Runtime [MBNJ09b], qui promeut les outils venus de l'IDM, non seulement à la conception mais aussi pour construire une abstraction exploitable en cours de fonctionnement. Cette approche permet ainsi un lien causal différé [MBJ+09a] entre l'abstraction et le système, introduisant ainsi un niveau de sûreté supérieur par la validation *a priori* du déploiement.

12.3 La divergence : outil pour la distribution

C'est dans le contexte de résolution des défis de conception et développement de systèmes adaptatifs distribués qu'interviennent les travaux de cette thèse, et ceci inclut le besoin de construire une abstraction pour leur communication mais également leur synchronisation. En effet, pour la réalisation de leur (auto-)adaptation, les DAS nécessitent une couche de réflexion qui leur permet, par introspection, de connaître leur état mais surtout, par intercession, de propager des mises à jour. Outre le besoin de modularité [Kru92] qui est issu à la fois des logiciels embarqués [Lee06a] mais également des systèmes large échelle [NFG+06], l'usage commun de distribution et d'adaptation dans ces systèmes induit de manière inévitable un problème de divergence des nœuds de calculs, et notamment de leurs couches de réflexion. Le problème de cette divergence a déjà été identifié pour les systèmes ultra large échelle [NFG+06] mais c'est également l'outil exploité par les algorithmes distribués afin de répartir et d'accéder de façon efficace à des données distribuées sur de nombreux nœuds [NRNK10] ,[KvS07], [BR02]. De manière analogue aux communications entre processus distants [GF99], [Gue99], les capacités de synchronisation des nœuds d'un DDAS sont hétérogènes car dépendantes des propriétés attendues en terme de cohérence, elles-mêmes dépendantes du cas d'usage. L'introduction d'un mécanisme de cohérence à terme [GA02], [BGL+06] dans la couche de réflexion permet par exemple des propagations de mises à jour dans des DDAS exploitant des réseaux maillés mobiles sporadiques.

12.4 Utiliser le Model@Runtime comme couche de réflexion de DDAS

Une abstraction doit permettre d'expliciter et d'aider à la manipulation d'un problème et non masquer sa complexité [GF99], [Gue99]. Cette affirmation est d'autant plus vraie dans les systèmes distribués qui, soumis à des communications distantes, doivent adapter cette complexité en fonction des propriétés à attendre, notamment en termes de cohérence de données et de systèmes. C'est en acceptant cette affirmation qu'est conçu le modèle proposé dans cette thèse, qui vise à offrir une abstraction commune aux différentes étapes de développement et du déploiement d'un DDAS. Ce modèle permet à la fois de modéliser la variabilité des nœuds de calcul mais également celle de ses capacités de synchronisation et de communication entre ses différents constituants. En étendant la notion de composant avec des fonctionnalités telles que les groupes et les

channels, ce modèle répond aux besoins de modularité en séparant les préoccupations et en permettant la construction de systèmes hétérogènes tant sur les communications mises en oeuvre que sur ces propriétés de cohérence. L'approche proposée exploite le Model@Runtime pour construire une couche de réflexion pour DDAS dont les propriétés de cohérence sont multiples et peuvent par extension prendre la forme d'une cohérence à terme. Au lieu de cacher la complexité à la fois des communications et de la synchronisation réseaux, ce modèle propose d'expliciter tant l'usage que la nature, pour permettre au concepteur d'exploiter ces notions afin de composer les propriétés à attendre de son DDAS.

12.5 Kevoree : une approche générique, expressive et performante du Model@Runtime distribué

Ce modèle nommé Kevoree a été implanté sous la forme d'un projet open source [1] afin de permettre son usage et sa validation sur différents types de DDAS. Au travers des différentes évaluations proposées dans ce document, il a été démontré que ce type d'approche permettait bien de modéliser et d'adapter des nœuds aussi hétérogènes que ceux trouvés dans les architectures basse consommation des CPS. Sa capacité à modéliser et à déployer différentes stratégies de synchronisation d'un même DDAS a été elle aussi démontrée sur un cas aussi complexe que la consistance à terme appliquée à l'échelle non plus d'une donnée mais d'un système adaptatif complet. Enfin, dans un dernier axe, cette thèse a discuté la propriété d'abstraction simplificatrice des DDAS fournie par Kevoree en offrant une modélisation non pas seulement expressive mais compréhensible et efficace pour les concepteurs de ces systèmes.

12.6 Une modélisation générique des systèmes adaptatifs qui s'inscrit dans une convergence CPS et Cloud

Finalement, le modèle Kevoree proposé cherche à introduire la gestion dynamique d'architecture directement dans le cycle de développement donc plus globalement dans le cycle de vie des DDAS. Ceci a permis, lors des évaluations, la construction de systèmes réactifs permettant d'accorder leur fonctionnement et donc leur architecture en fonction du contexte. Ce modèle de développement opportuniste sur l'assemblage a été exploité pour des architectures proches de l'internet des Objets puis pour des architectures plus puissantes de type *Cloud*. A la fin de cette thèse, on note une forte convergence des mécanismes d'adaptation disponibles pour les CPS et les *Clouds* qui laisse présager une convergence de ce domaine. Cette supposition est d'une part renforcée par la volonté d'interconnexion de l'IoT et du *Cloud*, et d'autre part par le récent essor des structures de type *Cloud* hybride alliant capacité d'hébergement privé et public. Ces sujets font partie des perspectives de travaux envisagées à la suite de cette thèse et présentées dans la section suivante.

1. kevoree.org

Chapitre 13

Perspectives

Le projet Kevoree a été conçu pour correspondre à une définition générique et réutilisable des DDAS. Le caractère opportuniste du style de programmation d'architecture qui en découle est applicable à de nombreux cas d'applications. Ces perspectives d'applications ont été initiées ou envisagées dans des collaborations avec d'autres travaux pendant et je l'espère après cette thèse.

Ces sujets de perspectives possibles incluent tout d'abord le problème de génération continue du modèle d'architecture. Cette étape cruciale pour le calcul des adaptations doit se plier au nouveau cycle continu en exploitant des algorithmes génétiques 13.1.1 ou des tissages de fragments d'architecture connus, comme dans le projet européen Diva.

Une autre piste est d'évaluer la capacité de Kevoree à rendre les algorithmes *gossip* dynamiques, en exploitant des projets comme GossipKit afin de rapprocher les architectures à composants des algorithmes d'échanges de données.

Les systèmes issus de l'approche *cloud computing* sont également une très bonne illustration de DDAS. Un des axes principaux de perspective de Kevoree est l'utilisation du M@R pour la modélisation de l'architecture et de l'adaptation de ces systèmes.

Enfin des travaux ont été également initiés pour effectuer de l'adaptation d'interactions homme-machine ou encore de l'adaptation dirigée par des simulations stochastiques.

13.1 Génération continue d'architecture

La construction du modèle d'architecture est l'étape centrale du processus d'adaptation proposé dans l'approche Kevoree . De la capacité d'évaluation et d'amélioration du modèle courant dépend la propriété d'élasticité et d'adaptabilité que l'on associe au DDAS. Les outils de l'IDM tels que les *feature models* ou les lignes de produits apportent des solutions pour la construction initiale d'une configuration. Mais ces outils sont encore peu compatibles avec les approches de conception continue préconisées par le M@R. Ceci est encore plus marqué avec l'approche opportuniste des dissémination de modèles suivant une approche *gossip* comme celle de cette contribution. Les taux d'erreurs de mise à jour conduisent alors à trouver des solutions qui permettent de

construire et reconstruire des adaptations de manière continue en vue d'optimiser le modèle. Ces solutions de synthèse doivent fonctionner de manière incrémentale pour profiter au mieux de la dissémination opportuniste sur les réseaux pair-à-pair. Deux perspectives différentes sont envisagées : d'un côté en exploitant une exploration des solutions suivant un algorithme génétique 13.1.1, d'un autre côté en cherchant à modéliser les migrations types par des patrons tels que ceux définis par Schmidt *et al.* [SSRB00] .

13.1.1 Approche génétique pour la résolution multi-axiale

La catégorie des algorithmes génétiques appartient à la famille des algorithmes évolutionnistes. Le principe de base de cette approche est de faire évoluer une solution par bonds (mutations) successifs pour la faire converger vers une optimisation voulue. Principalement exploités dans des cas d'usage où la solution optimale est non calculable, les algorithmes génétiques s'inspirent des règles de sélection naturelle pour assurer une convergence de la solution. Ces algorithmes en reprennent d'ailleurs le vocabulaire, chaque solution d'une population est appelée chromosome et chaque variant est appelé gène. Ces cas d'usage non calculables sont principalement des cas où l'optimisation se fait suivant plusieurs axes d'évaluation potentiellement orthogonaux. Dans un cas multi-axial la solution optimale est donc très difficilement calculable, l'approximation offerte par le parcours génétique permet de trouver un équilibre entre tous les axes afin de converger vers une solution acceptable et non idéale.

On trouve de nombreux cas où l'optimisation des DDAS est multi-axiale, par exemple l'optimisation de la performance est nécessairement antinomique avec la consommation électrique. Ceci est également vrai pour l'optimisation de métrique de qualité de services qui peuvent être antinomiques, par exemple l'approche de Malek, Medvidovic *et al* [MMMR12] montrent un exemple d'une telle optimisation sur des modèles proches de l'architecture. Dans cet article, l'optimisation de la qualité de service en terme de latence est antinomique avec la dimension qualité en terme de sécurité, un algorithme génétique y est alors appliqué pour cette optimisation. Le parcours génétique semble donc particulièrement adapté pour l'optimisation incrémentale requise par le processus M@R, par extension une approche sur un algorithme Greedy serait également a évaluer. Durant cette thèse ont été initiés des travaux avec Johan Bourcier et Benoît Baudry, pour évaluer la robustesse d'une optimisation multi-axiale sur le modèle Kevoree . En cours de publication, l'approche présentée ici ne contient que des résultats préliminaires.

13.1.1.1 Cas d'étude : optimisation de réseaux de capteurs

Le cas d'étude motivant cette perspective de recherche est nommé *SmartForest* et correspond à la définition d'un système ubiquitaire pour la supervision et la surveillance d'une forêt. Un tel système se caractérise par la composition de multiples capteurs communiquant sans fil de proche en proche, afin de transmettre et synchroniser les valeurs de différents capteurs physiques (fumée, humidité et température). Ce réseau de terrain est exploité par deux cas d'usage : la surveillance des points de départ d'incendie ainsi que l'observation du développement des végétaux de la forêt. Un modèle de cette

forêt doit donc comporter des informations de géo-localisation ainsi que la topologie des capteurs de la forêt avec leurs équipements de mesure déployés.

Plusieurs critères définissent alors la qualité d'une telle configuration de forêt : la précision des données collectées (temps de collecte vis-à-vis du phénomène), la consommation énergétique de l'ensemble, la qualité de communication, le délai de propagation sur des points d'agrégation et de sauvegarde. Certains de ces critères sont alors en conflit direct : la qualité des données dépend de la fréquence d'échantillonnage des capteurs qui est elle-même liée à la consommation énergétique. Augmenter la qualité en réduisant la consommation est alors impossible, les architectes de tels systèmes doivent alors trouver un équilibre acceptable parmi toutes les solutions déployables.

Ce type de système doit inclure de l'adaptation pour sa propre survie. Par exemple lorsqu'une batterie d'un capteur devient faible il faut alors déplacer les capteurs logiciels vers un nœud physiquement proche et recalculer ainsi la *qualité globale* obtenue. Les configurations sont donc non fixes et évoluent durant le fonctionnement du réseau de capteur.

13.1.1.2 Évaluation du domaine d'exploration

Dans ce cas de supervision de forêt, le calcul du nombre de configurations se fait en prenant en compte le nombre de capteurs physiques déployés, le nombre et la position des composants logiciels ainsi que leurs paramètres. Dans notre cas où les capteurs physiques ont trois capteurs physiques connectés, le nombre de composants logiciels déployés peut varier de 0 à 3 (température, fumée et humidité). Chaque composant logiciel peut fonctionner à différentes fréquences d'échantillonnage : cinq valeurs de période de mesure sont disponibles sur ces capteurs logiciels.

Le nombre de configurations par nœud (capteur physique) est alors : $5^3 = 125$. Si on considère le nombre de nœuds disponibles n, ce nombre s'élève à 125^n pour le nombre de configurations pour la forêt globale.

De fait, si on prend une nombre très faible de capteurs on arrive approximativement à $2*10^8$ configurations possibles. Mais si on considère un réseau de capteurs plus réaliste de 400 capteurs physiques déployés, ce nombre s'élève à $5 * 10^{838}$. La recherche de la meilleure solution est par définition non réaliste avec les processeurs actuels. Le domaine d'exploration illustre parfaitement le besoin d'un équilibre entre tous ces critères pour trouver rapidement une solution acceptable sans pour autant explorer l'ensemble du domaine.

13.1.1.3 Approche : définition de mutation au niveau modèle

Là encore l'approche proposée pour résoudre ce problème classique de configuration de réseau de capteurs est d'exploiter une couche d'abstraction pour la manipulation du DDAS constitué par les éléments de la forêt. L'approche globale est donc l'exploitation d'un modèle Kevoree par état de la forêt pour en évaluer sa qualité. Les nœuds de capteurs physiques sont de la même nature que ceux décrits dans le chapitre validation micro-contrôleur8. Chaque nœud physique est donc représenté par un nœud Kevoree ,

l'ensemble des capteurs disponibles est défini par des *ComponentType* accompagnés de *ChannelType* pour la communication entre eux. L'allumage et l'extinction d'un capteur logiciel est donc représenté par le déploiement d'un composant instance Kevoree sur ce nœud. Le changement de valeur d'échantillonnage est représenté par un dictionnaire paramétrique sur les instances.

L'approche globale utilise le modèle Kevoree en entrée des *fitness function* qui servent à évaluer la qualité d'un modèle vis-à-vis d'un axe à optimiser, par exemple la consommation électrique. Les opérations de mutation sont alors des transformations de modèles qui ajoutent et suppriment des composants instances ou modifient des valeurs de dictionnaire. Pour simplifier le développement de ces opérateurs de mutation, une abstraction de patrons d'évolution est proposée au niveau modèle. Ces patrons exploitent une approche par tissage de script d'évolution pour modifier le modèle dans des proportions prévues. Par exemple un script peut prévoir d'ajouter un composant de température, ou de supprimer une instance, ou encore de faire baisser la fréquence d'échantillonnage d'une instance. Le but de ces scripts est d'exprimer les évolutions avec et au niveau modèle pour rester cohérent avec l'abstraction offerte pour le développement d'un tel système.

Ainsi dans la nouvelle terminologie, un individu est un modèle Kevoree , une population un ensemble de ces modèles, un gène est un composant instance, une opération de mutation est un patron d'évolution et une *fitness fonction* est une évaluation d'un modèle Kevoree selon un axe.

13.1.1.4 Résultats préliminaires

Ces résultats sont encore en cours de publication, mais les résultats préliminaires montrent que ce type d'approche fait converger une forêt de 256 capteurs vers une solution proche de l'optimal en moins de 90 secondes sur une machine dotée d'un processeur dual-core cadencé à 3 Ghz. Le type de donnée en entrée a cependant une influence importante sur la vitesse de convergence ainsi que sur la qualité des patrons d'adaptation. Cette piste de perspective semble très adaptée à l'usage à l'exécution puisque ces temps d'exécution sont compatibles avec les temps d'adaptation demandés dans ces cas d'usage.

13.1.2 Fragment / Aspect / Patron d'architecture

Les approches par patrons ont déjà été exploitées avec succès par exemple pour la programmation par flux avec les patrons d'architectures de Schmitt *et al* ou avec les EIP. Se pose alors la question : quel est le meilleur niveau d'abstraction pour raisonner et pour quel but ? La notion de composant encapsule bien les fonctionnalités ainsi que le déploiement d'un système, cependant les patrons d'architecture permettent d'expliciter des styles d'architectures connues avec des propriétés associées. Par exemple un patron *load balancer* regroupe plusieurs composants et est associé à une baisse de charge sur les ports réseaux de réponse. L'approche par patrons permet alors d'anticiper des types d'adaptation pour ainsi simplifier l'écriture de cas d'évolution d'un modèle de DDAS.

Ce type d'approche est complémentaire de l'approche de parcours génétiques. Un patron d'évolution tel qu'envisagé pour le modèle Kevoree est donc composé de deux parties : un point de coupe, un script d'évolution exploitant le point de coupe pour altérer le modèle. Proche de la programmation par aspects telle qu'elle a été proposée par le modèle du projet européen Diva, une perspective d'évolution du modèle Kevoree est donc la définition d'un langage de patron de niveau architecture pour son exploitation par les raisonneurs.

13.1.3 Utilisation continu de modèle de contexte

La modélisation du système proposé dans Kevoree couvre essentiellement les aspects structurels du système. Un modèle de métrique est également inclus dans cette structure mais il ne couvre que les éléments primaires du système, à savoir la charge mémoire, CPU, etc. Or pour prendre des décisions de reconfiguration de nombreux travaux essentiellement utilisés au moment de la conception du système ont développé des modèles plus élaborés permettant de représenter de façon plus fine les variations du système.

Ainsi en couplant les informations provenant d'un *monitoring* du système à des informations extra-fonctionnelles que l'on peut ajouter en décoration du modèle structurel, il est alors possible de faire de la prévision sur le comportement du système. Les travaux autour du modèle de composant Palladio [BKR09],[BKR07] visent justement à proposer de telles techniques au moment de la conception pour anticiper les performances du système suivant les configurations de celui-ci.

L'approche Model@Runtime et la notion de conception continue (*continuous design*) pourraient largement profiter d'une extension du modèle structurel pour se servir des modèles d'anticipation de performance tels qu'il seraient définis dans Palladio. Ainsi les variations du système en termes de métriques pourraient être mises en corrélation de façon continue pour à la fois surveiller le système vis-à-vis des performances annoncées par le modèle mais aussi le modèle théorique lui-même. En effet si le modèle ne réagit plus comme le système théorique servant à la prévision, il serait alors intéressant de le remettre en cause afin de l'améliorer pour obtenir une simulation de plus en plus fiable.

En d'autres termes, une perspective de poursuite sur ce sujet serait de travailler avec l'équipe qui développe le modèle Palladio pour rendre le système de prévision de performance exploitable en conception continue au dessus de Kevoree . En associant cela avec une fonction d'apprentissage les résultats de tels systèmes pourraient être transférés et réinjectés dans le modèle Kevoree pour les systèmes *cloud* détaillés ci-dessous.

13.2 Modéliser la dynamicité des protocoles eux-mêmes

Les protocoles exploités dans les réseaux pair-à-pair connaissent de nombreux points de variation qui donnent lieu à de nombreuses dérivations de ces algorithmes. Ainsi chaque variation de *gossip* offre des propriétés différentes intéressantes suivant les cas d'usage, par exemple en passant d'un mode de communication *pull* à un mode *push/pull*. Ce besoin est tel que des projets tels que GossipKit [LTB08],[Tai10],[LTB+07] cherchent à exprimer la configuration de tels algorithmes à l'aide de composants et d'injection de

dépendances. Sur bien des points les besoins de GossipKit se rapprochent du M@R, l'architecture à composants pourrait alors servir de support pour les architectures de GossipKit et surtout son déploiement sur les différents noeuds.

En extrapolant encore cette perspective, rapprocher le modèle et son utilisation @Runtime et les protocoles *gossip* poussent à réfléchir à l'adaptation dynamique de ces protocoles. En effet si le M@R peut dans un premier temps être exploité pour le déploiement ou comme couche de réflexion par exemple pour la topologie réseau, dans un deuxième temps sa capacité de divergence peut être à exploiter. En effet le caractère opportuniste des propagations *gossip* pourrait être appliqué à son propre fonctionnement *via* cette approche. En d'autres termes les agents participant à ces échanges pourraient chacun décider de mise à jour pour adapter le protocole et ainsi optimiser son fonctionnement. À cause de la divergence du M@R il faudrait alors considérer la divergence des différents agents du *gossip*. Sur bien des points encore ces besoins se rapprochent des attaques byzantines et pourraient réutiliser ces résultats pour rendre les protocoles *gossip* résistants à l'adaptation dynamique . Outre la vision d'architecture que cette approche apporterait, les échanges pourraient profiter des mêmes règles et outils pour générer des adaptations d'optimisation opportunistes que les DDAS.

13.3 Apprentissage d'erreur opportuniste

La dissémination opportuniste proposée par le modèle Kevoree pose une véritable question pour la reprise d'erreur. En effet si ce problème a été traité pour les protocoles de type consensus, il reste entier pour des propagations de type *gossip*. Une piste serait d'exploiter de manière opportuniste les informations d'échec de déploiement. Couplé à un moteur d'apprentissage il serait alors envisageable d'apprendre et d'avoir un retour sur la dissémination et par exemple détecter des déploiements infectieux avant qu'ils n'atteignent la totalité du DDAS. Ainsi à la manière d'une pandémie bactérienne il doit être possible de vérifier l'état des nœuds après *infection* par le nouveau modèle. En cas de pertes trop importantes ce type d'apprentissage pourrait détecter des déploiements problématiques et anticiper leur annulation.

13.4 Kevoree pour le *cloud computing*

L'utilisation du modèle Kevoree pour la gestion du *cloud computing* dans la collaboration avec Erwan Daubert s'est avérée très prometteuse. L'usage du Model@Runtime dans ce type de contexte permet d'assurer une réutilisation possible des algorithmes d'élasticité et surtout permet d'aller au-delà de l'adaptation d'un simple *cloud* puisque la notion de nœud hétérogène permet d'envisager des interactions et adaptations entre *clouds*. L'assemblage de ces infrastructures, nommé *sky*, nécessite des adaptations le plus souvent pour des raisons économiques. Il est en effet nécessaire de faire migrer des composants logiciels d'une infrastructure à une autre pour réduire non plus la charge logicielle mais le coût économique. La poursuite de ces travaux de modélisation de *cloud* est envisagée au travers de plusieurs projets européens dont notamment le projet *open*

source CloudML en collaboration avec la structure SINTEF en Norvège.

Vers des *clouds* (*sky*) hybrides, mélangeant *data center* et hébergement personnel Les travaux autour du Raspberry Pi et de l'intégration avec les capteurs dans le cadre du projet Entimid, font ressortir un certain intérêt pour l'hébergement de logiciels au plus proche des utilisateurs et donc dans l'habitat personnel. En ajoutant à cela le problème économique des coûts d'hébergement, il est envisageable que des solutions de type *cloud* hybrides voient le jour d'ici peu. En substance ce type de *cloud* va mélanger des hébergements de logiciels en partie dans un nœud de calcul local et en partie dans un nœud distant. La problématique de l'adaptation intervient lorsqu'il faut déplacer des composants d'un nœud infrastructure vers le nœud local et inversement. L'avantage de ce système est de pouvoir déployer du logiciel au plus près des utilisateurs en cas de besoin et de pouvoir le migrer dans une infrastructure, par exemple en cas de panne ou de déplacement. Les plates-formes comme le Raspberry PI permettent d'envisager des hébergements à très faible coût en terme de consommation électrique et donc tout à fait déployable de façon locale si une solution de sauvegarde liée avec un *cloud* est prévue.

Plusieurs collaborations entre le laboratoire Irisa/INRIA de Rennes, le SINTEF en Norvège ainsi que le SnT au Luxembourg sont envisagées pour poursuivre ces travaux.

13.5 Adaptation d'interface : exploiter Kevoree dans un naviguateur Web

Récemment les navigateurs Web sont devenus des plates-formes d'exécution très versatiles et dynamiques. On voit donc fleurir chez la fondation Mozilla ou chez le navigateur Chrome des plates-formes d'application sur étagère. De plus les environnements Javascript intégrés dans les navigateurs et notamment avec les avancées du HTML5 permettent d'envisager des interfaces utilisateurs très dynamiques reprenant le plus souvent beaucoup de composants visuels communs.

Ces environnements dynamiques seraient donc une cible particulièrement adaptée pour l'exécution d'une approche Model@Runtime permettant la réalisation d'un interaction homme-machine adaptative côté client. Une perspective de développement de Kevoree est donc d'assurer une compatibilité de l'implantation avec ces environnements d'exécution et d'imaginer représenter les clients Web comme des conteneurs de composants liés à des composants serveurs.

13.6 Couplage modèle comportemental et modèle d'architecture

L'approche préconisée par le modèle Kevoree se résume essentiellement à une vision comportementale des DDAS. Cette vision peut se voir accompagnée de différentes informations qui s'organisent sous la forme de modèles périphériques. Ces décorations

de modèle peuvent ajouter des informations de contexte telles qu'un modèle de métriques de fonctionnement. Ce modèle héberge par exemple les temps de réponse d'un port pendant un fonctionnement. Mais au-delà il est également possible de décorer le modèle structurel des définitions de type avec des informations comportementales. Ces informations peuvent être par la suite exploitées pour anticiper des comportements de composants, et ainsi par exemple calculer le temps théorique du temps de réponse d'un port. Les deux approches sont conjugables pour faire de la vérification *a priori* et *a posteriori* des performances.

Une perspective d'évolution de Kevoree est la définition de ce modèle, ce lien a déjà été commencé grâce aux liens avec les travaux de thèse de Viet-Hoa Nguyen. Ces travaux ont déjà donné lieu à une publication au séminaire MRT 2012 associé à la conférence MODELS'2012 [NFPB12]. Dans ces travaux une couche comportementale est ajoutée au modèle Kevoree pour décrire le comportement théorique des *TypeDefinition* à l'aide d'un modèle stochastique fondé sur une description à l'aide de modèle de réseaux de Petri [Pet81]. L'objectif de ces travaux est d'utiliser ce modèle comportemental à l'exécution pour faire de l'anticipation de performance.

De ces travaux découle le besoin fort de définition de langage permettant d'assurer une cohérence entre ce modèle comportemental et le modèle structurel extrait du code source réel. Cette perspective ouvre la voie vers un usage à l'exécution des modèles de simulation actuellement employés pour la simulation de logiciel. Ce nouvel usage conjointement associé à un *monitoring* permettrait d'aider les raisonneurs afin de mieux anticiper les effets des adaptations des DDAS.

13.7 Chargement de code à chaud et sécurité des plates-formes Java et Dalvik

Le développement de KevoreeClassLoader et le chargement de code à chaud en général posent de vraies questions de sécurité. En effet beaucoup d'approches telles que celle de Bartel *et al* [BKLTM12] emploient l'analyse statique de code afin de vérifier le type de ressources utilisées par exemple dans un *smart phone*. En autorisant le chargement de code à chaud , un environnement tel qu'un *smart phone* Android s'expose à ce qu'une application charge du code potentiellement malicieux.

De façon plus générale sur les machines virtuelles Java et dérivées, il est très difficile de faire ce que l'on pourrait appeler du *micro-sandboxing*, en d'autres termes l'isolation d'un processus vis -à-vis de certaines ressources, comme la création de processus ou de *socket* réseau. Cette limitation fait qu'il est toujours actuellement difficile d'assurer l'isolation entre deux parties de logiciel Java et donc par extension entre deux composants disponibles sur la même machine virtuelle.

Le modèle de composant Kevoree définit pourtant une isolation des processus de chaque instance, il serait alors possible de piloter une isolation de leurs ressources au niveau machine virtuelle si les primitives de bas niveau étaient disponibles.

Il est envisagé comme perspective à ces travaux de réfléchir à ces deux problèmes dans le projet InfraJVM, qui a commencé à Rennes cette année. Le but de ce projet est

d'apporter des réponses au niveau environnement d'exécution tant d'un point de vue règle de sécurité et d'isolation que de ressources.

Bibliographie

[ABP+03] F. André, J. Buisson, J.L. Pazat, et al. Dynamic adaptation of parallel and distributed components on grid environments. 2003.

[ABP05] Françoise André, Jérémy Buisson, and Jean-Louis Pazat. Dynamic adaptation of parallel codes : Toward self-adaptable components for the grid, 2005.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. Archjava : connecting software architecture to implementation. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 187–197. IEEE, 2002.

[ACN06] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in archjava. *ECOOP 2002—Object-Oriented Programming*, pages 185–193, 2006.

[Agh85] G.A. Agha. Actors : a model of concurrent computation in distributed systems. 1985.

[AK03a] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36 – 41, sept.-oct. 2003.

[AK03b] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, 2003.

[All03] O.S.G. Alliance. *Osgi service platform, release 3*. IOS Press, Inc., 2003.

[APW+08] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG*, volume 2. Prentice Hall, 1996.

[BADI05] H. Blohm, SAP AG, J.J. Dubray, and A.C. Interface21. Service component architecture. 2005.

[Bar05] O. Barais. Construire et maıtriser l'evolution d'une architecture logicielle a base de composants. *These de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Lille, France*, 2005.

[BB72] Geneviève Boulinier and Georges Boulinier. Les polynésiens et la navigation astronomique. *Journal de la Société des océanistes*, 28(36) :275–284, 1972.

[BB06] N. Bencomo and G. Blair. Genie : a domain-specific modeling tool for the generation of adaptive and reflective middleware families. In *6th OOPSLA Workshop on Domain-Specific Modeling*. Citeseer, 2006.

[BBF07] N. Bencomo, G. Blair, and R. France. Summary of the workshop models@ run. time at models 2006. *Models in Software Engineering*, pages 227–231, 2007.

[BBF09a] G. Blair, N. Bencomo, and R.B. France. Models@ run. time. *Computer*, 42(10) :22–27, 2009.

[BBF09b] G. Blair, N. Bencomo, and R.B. France. Models@ run.time. *Computer*, 42(10) :22 –27, oct. 2009.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. Ieee, 2006.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 303–311. ACM, 1990.

[BCBB11] Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981, pages 62–76, Wellington, Nouvelle-Zélande, 2011. Springer Berlin / Heidelberg.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36 :1257–1284, September 2006.

[BCS06] M. Baker, B. Carpenter, and A. Shafi. Mpj express : towards thread safe java hpc. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[Beu00] A. Beugnard. Communication services as components for telecommunication applications. In *Proc. 14th European Conference on Object-Oriented Programming, ECOOP'2000*. Citeseer, 2000.

[BF97] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2) :281–300, 1997.

[BGF+08] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 811–814. IEEE, 2008.

[BGL+06] R. Baldoni, R. Guerraoui, R. Levy, V. Quema, and S. Piergiovanni. Unconscious eventual consistency with gossips. *Stabilization, Safety, and Security of Distributed Systems*, pages 65–81, 2006.

[BGW93] D.G. Bobrow, R.P. Gabriel, and J.L. White. Clos in context-the shape of the design space. *Object Oriented Programming : The CLOS Perspective*, pages 29–61, 1993.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48. IEEE, 2006.

[BHP+07] T. Bures, P. Hnetynka, F. Plasil, J. Klesnil, O. Kmoch, T. Kohan, and P. Kotrc. Runtime support for advanced component concepts. In *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 337–345. IEEE, 2007.

[BJ02] D. Brandon Jr. Crud matrices for detailed object oriented design. *Journal of Computing Sciences in Colleges*, 18(2) :306–322, 2002.

[BJPW99] A. Beugnard, J.M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[BKLTM12] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler : converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[BKR07] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM, 2007.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1) :3–22, 2009.

[BPR99] F. Bellifemine, A. Poggi, and G. Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[BR02] R. Baldoni and Raynal. Fundamentals of distributed computing. *IEEE Distributed Systems Online*, 3(2) :1–18, 2002.

[Bre00] E.A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[BSBG08] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically adaptive systems are product lines too : Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland*, volume 38, page 40, 2008.

[CAS08] A.A. Cardenas, S. Amin, and S. Sastry. Secure control : Towards survivable cyber-physical systems. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 495–500. IEEE, 2008.

[CCSV07] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A classification framework for component models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, October 2007.

[CGFP09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime : The case of smart homes. *Computer*, 42(10) :37–43, 2009.

[CGS+02] S.W. Cheng, D. Garlan, B. Schmerl, J.P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for system self-repair. 2002.

[Cha87] A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3) :350–366, 1987.

[CHG+04] S.W. Cheng, A.C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. An architecture for coordinating multiple self-management systems. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 243–252. IEEE, 2004.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10) :667–668, 1971.

[CL98] M. Castro and B. Liskov. Practical byzantine fault tolerance. *Operating Systems Review*, 33 :173–186, 1998.

[CL02] I. Crnkovic and M.P.H. Larsson. *Building reliable component-based software systems*. Artech House Publishers, 2002.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2001.

[CRKGLA89] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *SIGCOMM Comput. Commun. Rev.*, 19 :224–236, August 1989.

[DBE08] A. Diaconescu, J. Bourcier, and C. Escoffier. Autonomic ipojo : Towards self-managing middleware for ubiquitous systems. In *Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing,*, oct. 2008.

[DG08] J. Dean and S. Ghemawat. Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113, 2008.

[DGR09] D. Donsez, K. Gama, and W. Rudametkin. Developing adaptable components using dynamic languages. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 396–403. IEEE, 2009.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1(2) :115–138, 1971.

[Dij01] E.W. Dijkstra. The structure of the" the"-multiprogramming. *Classic Operating Systems : From Batch Processing to Distributed Systems*, page 223, 2001.

[DL+06] P.C. David, T. Ledoux, et al. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP*, volume 6, 2006.

[DSH+09] J. Domaschka, H. Schmidt, F.J. Hauck, R. Kapitza, and H.P. Reiser. Dosgi : An architecture for instant replication. In *Supplement Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, 2009.

[DWH07] Tom De Wolf and Tom Holvoet. Design patterns for decentralised coordination in self-organising emergent systems. In Sven Brueckner, Salima Hassas, Márk Jelasity, and Daniel Yamins, editors, *Engineering Self-Organising Systems*, volume 4335 of *Lecture Notes in Computer Science*, pages 28–49. Springer Berlin / Heidelberg, 2007.

[EDSMG10] Jacky Estublier, Idrissa A. Dieng, Eric Simon, and Diana Moreno-Garcia. Opportunistic computing experience with the sam platform. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '10, pages 1–7, New York, NY, USA, 2010. ACM.

[EGKM04] P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE computer*, 37(5) :60–67, 2004.

[EH07] C. Escoffier and R. Hall. Dynamically adaptable applications with ipojo service components. In *Software Composition*, pages 113–128. Springer, 2007.

[EHL07] C. Escoffier, R.S. Hall, and P. Lalanda. ipojo : an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481. IEEE, 2007.

[Ehr10] D. Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[ERRJ95] G. Erich, H. Richard, J. Ralph, and V. John. Design patterns : elements of reusable object-oriented software. *Reading : Addison Wesley Publishing Company*, 1995.

[FBP+12] François Fouquet, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel, Brice Morin, and Franck Fleurey. A Dynamic Component Model for Cyber Physical Systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, Bertinoro, Italie, July 2012.

[FDB+09] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.M. Jézéquel. Modeling and validating dynamic adaptation. *Models in Software Engineering*, pages 97–108, 2009.

[FDP+12a] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, A. Blouin, et al. Kevoree : une approche model@ runtime pour les systèmes ubiquitaires. In *UbiMob2012*, 2012.

[FDP+12b] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *DAIS 2012*, Stockholm, Suède, June 2012.

[FG97] S. Franklin and A. Graesser. Is it an agent, or just a program ? : A taxonomy for autonomous agents. *Intelligent Agents III Agent Theories, Architectures, and Languages*, pages 21–35, 1997.

[FHS+06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2) :62–70, 2006.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th ACSC*, volume 10, pages 56–66, 1988.

[Fip97] F. Fipa. specification part 2 : Agent communication language. Technical report, Technical report, FIPA-Foundation for Intelligent Physical Agents, 1997.

[FKN+92] A. FINKELSTEIN, J. KRAMER, B. NUSEIBEH, L. FINKELSTEIN, and M. GOEDICKE. Viewpoints : A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 02(01) :31–57, 1992.

[FL03] M. Fox and D. Long. Pddl2.1 : An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20 :61–124, 2003.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2) :374–382, 1985.

[FMG02] L. Fiege, G. Mühl, and F.C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(04) :359–388, 2002.

[FNM+12] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Models 2012*, Innsbruck, Autriche, October 2012.

[Fow04] M. Fowler. Inversion of control containers and the dependency injection pattern, 2004.

[G+05] J.J. Garrett et al. Ajax : A new approach to web applications. 2005.

[GA02] S. Gustavsson and S.F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems*, pages 105–107. ACM, 2002.

[Gam95] E. Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gar10] Gartner. Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010 ; smartphone sales increased 96 percent, 2010. http ://www.gartner.com/it/page.jsp ?id=1466313.

[GC03] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18, 2003.

[GCH+04] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004.

[GCS03] D. Garlan, S.W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, pages 61–89, 2003.

[GF99] R. Guerraoui and M.E. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4) :101–104, 1999.

[GL02] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2) :51–59, 2002.

[GLVB+03] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language : A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[GME07] D. Goodman, M. Morrison, and B. Eich. *Javascript® bible*. John Wiley & Sons, Inc., 2007.

[GMK02] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM, 2002.

[Gol89] D.E. Goldberg. Genetic algorithms in search, optimization, and machine learning. 1989.

[GS99] A.S. Gokhale and D.C. Schmidt. Optimizing a corba internet inter-orb protocol (iiop) engine for minimal footprint embedded multimedia systems. *Selected Areas in Communications, IEEE Journal on*, 17(9) :1673–1706, 1999.

[GSTV11] D. Ghosh, J. Sheehy, K.K. Thorup, and S. Vinoski. Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications*, pages 1–8, 2011.

[GSTV12] D. Ghosh, J. Sheehy, K.K. Thorup, and S. Vinoski. Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications*, pages 1–8, 2012.

[GT04] John C. Georgas and Richard N. Taylor. Towards a knowledge-based approach to architectural adaptation management. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, pages 59–63, New York, NY, USA, 2004. ACM.

[Gue99] R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. *Informatik*, 2 :3–8, 1999.

[har] The Internet of Things Meets The Internet of People. `http://www.harborresearch.com/_literature_60961/The_Internet_of_Things_Meets_The_Internet_of_People`.

[HC01] G.T. Heineman and W.T. Councill. *Component-based software engineering : putting the pieces together*, volume 17. Addison-Wesley USA, 2001.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[HFP02] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. *Distributed Computing*, pages 265–279, 2002.

[HKJR10] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed. Zookeeper : Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[HO09] P. Haller and M. Odersky. Scala actors : Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3) :202–220, 2009.

[Hoa74] C.A.R. Hoare. Monitors : An operating system structuring concept. *Communications of the ACM*, 17(10) :549–557, 1974.

[HP06] P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. *Component-Based Software Engineering*, pages 352–359, 2006.

[HW04] G. Hohpe and B. Woolf. *Enterprise integration patterns : Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[Jac02] D. Jackson. Alloy : a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2) :256–290, 2002.

[JB06a] M. Jelasity and O. Babaoglu. T-man : Gossip-based overlay topology management. *Engineering Self-Organising Systems*, pages 1–15, 2006.

[JB06b] S. Jin and A. Bestavros. Small-world characteristics of internet topologies and implications on multicast scaling. *Computer Networks*, 50(5) :648–666, 2006.

[Jen00] N.R. Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2) :277–296, 2000.

[JK06] M. Jelasity and A.M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on*, pages 117–124. IEEE, 2006.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3) :219–252, August 2005.

[JRL08] N. Jussien, G. Rochart, and X. Lorca. The choco constraint programming solver. In *CPAIOR08 workshop on OpenSource Software for Integer and Contraint Programming OSSICP08*, 2008.

[JVG+07] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3) :8, 2007.

[JVM05] W.L. Johnson, H. Vilhjalmsson, and S. Marsella. Serious games for language learning : How much game, how much ai. *Artificial Intelligence in Education : Supporting Learning through Intelligent and Socially Informed Technology*, pages 306–313, 2005.

[JW97] Ralph Johnson and Bobby Woolf. The type object pattern, 1997.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, January 2003.

[Keo03] J.E. Keogh. J2me : The complete reference. 2003.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.

[KKR+12] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low power or high performance ? a tradeoff whose time has come (and nearly gone), 2012.

[Kru92] C.W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2) :131–183, 1992.

[KvS07] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5) :2–7, October 2007.

[KWB03] A.G. Kleppe, J.B. Warmer, and W. Bast. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[Lam01] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4) :18–25, 2001.

[Lee06a] E.A. Lee. Cyber-physical systems-are computing foundations adequate. In *Position Paper for NSF Workshop On Cyber-Physical Systems : Research Motivation, Techniques and Roadmap*, volume 2. Citeseer, 2006.

[Lee06b] E.A. Lee. The problem with threads. *Computer*, 39(5) :33 – 42, may 2006.

[Lee08] E.A. Lee. Cyber physical systems : Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[LLC07] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *ARM '07 : Proc of the 6th international workshop on Adaptive and reflective middleware*, pages 1–6, Newport Beach, CA, 2007. ACM.

[LTB+07] S. Lin, F. Taiani, G.S. Blair, et al. Gossipkit : A framework of gossip protocol family. In *Proceedings of the 5th MiNEMA Workshop (Middleware for Network Eccentric and Mobile Applications)*, pages 26–30, 2007.

[LTB08] Shen Lin, François Taïani, and Gordon S. Blair. Facilitating gossip programming with the gossipkit framework. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 238–252, Berlin, Heidelberg, 2008. Springer-Verlag.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[MB05] Selma Matougui and Antoine Beugnard. Two ways of implementing software connections among distributed components, 2005.

[MBJ+09a] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10) :44 –51, oct. 2009.

[MBJ+09b] B. Morin, O. Barais, J.M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10) :44–51, 2009.

[MBNJ09a] B. Morin, O. Barais, G. Nain, and J.M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[MBNJ09b] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In

Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[MGH+98] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl-the planning domain definition language. 1998.

[mis] Human reaction times as a response to delays in control systems. `http://www.measurepolis.fi/alma/ALMA%20Human%20Reaction%20Times%20as%20a%20Response%20to%20Delays%20in%20Control%20Systems.pdf`.

[MMMR12] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1) :73–100, 2012.

[MMR+10] R. Mélisson, P. Merle, D. Romero, R. Rouvoy, and L. Seinturier. Reconfigurable run-time support for distributed service component architectures. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 171–172. ACM, 2010.

[Mon08] Alberto Montresor. Intelligent gossip, 2008.

[Mor10a] B. Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability.* 2010.

[Mor10b] Brice Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability.* PhD thesis, Université Rennes 1, Septembre 2010.

[MR10] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture).* Addison-Wesley Professional, 2010.

[MSKC04] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7) :56–64, 2004.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1) :70–93, 2000.

[MZ08] A. Montresor and R. Zandonati. Absolute slicing in peer-to-peer systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008.

[Nai11] G. Nain. *EnTiMid : Un modele de composants pour integrer des objets communicants dans des applications a base de services.* PhD thesis, Universite Rennes 1, 2011.

[NBF+09] G. Nain, O. Barais, R. Fleurquin, J.M. Jezequel, et al. Entimid : un middleware aux services de la maison. In *RNTI L 4 CAL 2009 (3e Conference francophone sur les Architectures Logicielles)*, pages 59–72, 2009.

[NDBJ08a] G. Nain, E. Daubert, O. Barais, and J.M. Jezequel. Using mde to build a schizophrenic middleware for home/building automation. *Towards a Service-Based Internet*, pages 49–61, 2008.

[NDBJ08b] Grégory Nain, Erwan Daubert, Olivier Barais, and Jean-Marc Jézéquel. Using mde to build a schizophrenic middleware for home/building automation. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, ServiceWave '08, pages 49–61, Berlin, Heidelberg, 2008. Springer-Verlag.

[NFG+06] L. Northrop, P. Feiler, R.P. Gabriel, J. Goodenough, R. Linger, R. Kazman, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems-the software challenge of the future. *Technical report Software Engineering Institute Carnegie Mellon University ISBN*, 2006.

[NFM+10] G. Nain, F. Fouquet, B. Morin, O. Barais, and J.M. Jezequel. Integrating iot and ios with a component-based approach. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 191–198. IEEE, 2010.

[NFPB12] Viet Hoa Nguyen, François Fouquet, Noël Plouzeau, and Olivier Barais. A Process for Continuous Validation of Self-Adapting Component Based Systems. In *7th International Workshop on Models@run.time of the MODELS 2012 Conference.*, Innsbruck, Autriche, December 2012.

[NRNK10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4 : Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, dec. 2010.

[OGT+99] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3) :54–62, 1999.

[OMG11] OMG. Omg's metaobject facility, 2011. `http://www.omg.org/mof`.

[PCBD10] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. *Feature-Based Composition of Software Architectures*, volume 6285 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[Pet81] J.L. Peterson. Petri net theory and the modeling of systems. *PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632, 1981, 290*, 1981.

[PHP87] D. Pilaud, N. Halbwachs, and JA Plaice. Lustre : A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[PKBGS08] An Phung-Khac, Antoine Beugnard, Jean-Marie Gilliot, and Maria-Teresa Segarra. Model-driven development of component-based adap-

tive distributed applications. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2186–2191, New York, NY, USA, 2008. ACM.

[Pri08] D. Pritchett. Base : An acid alternative. *Queue*, 6(3) :48–55, 2008.

[PSMR11] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the barrelfish os and the single-chip cloud computer. In *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC), Ettlingen, Germany*, 2011.

[RAY12] M. RAYNAL. Concurrent programming. 2012.

[RBD+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. *MUSIC : Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments*, volume 5525 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009.

[RHP+03] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.M. Kermarrec. Adaptive gossip-based broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 47–56, 2003.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. *ECOOP'98—Object-Oriented Programming*, pages 550–570, 1998.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF : eclipse modeling framework*. Addison-Wesley Professional, 2008.

[SFR04] W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network Programming : The Sockets Networking API*, volume 1. Addison-Wesley Professional, 2004.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component software : beyond object-oriented programming*. Addison-Wesley, 2002.

[Sim11] E. Simon. *Sam : un environnement d'exécution pour les applications à services dynamiques et hétérogènes*. PhD thesis, Université de Grenoble, 2011.

[SMF+09a] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.B. Stefani. Reconfigurable sca applications with the frascati platform. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 268–275. IEEE, 2009.

[SMF+09b] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Bangalore Inde, 2009. IEEE. IST FP7 IP SOA4All.

[SMK11] D. Sykes, J. Magee, and J. Kramer. Flashmob : distributed adaptive self-assembly. In *Proceeding of the 6th SEAMS*, pages 100–109. ACM, 2011.

[SMR+11] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5) :559–583, 2011.

[SPDC06] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. *Component-Based Software Engineering*, pages 139–153, 2006.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture : Patterns for concurrent and networked objects*, volume 2. Wiley, 2000.

[Sto09] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE '09.*, pages 369 –374, aug. 2009.

[Sut05] H. Sutter. The free lunch is over : A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3) :202–210, 2005.

[Syk10] D. Sykes. Autonomous architectural assembly and adaptation. *PhD in computing, Imperial College of Science, Technology and Medicine-Department of Computing*, 2010.

[Tai10] Francois Taiani. Gossipkit web site, 2010. http ://f-taiani.ouvaton.org/GossipKit.

[TGP08] Y. Tan, S. Goddard, and L.C. Perez. A prototype architecture for cyber-physical systems. *ACM SIGBED Review*, 5(1) :1–2, 2008.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2) :180–209, June 1979.

[TTP+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5) :172–182, 1995.

[Tur97] J. Turley. Atmel avr brings risc to 8-bit world. *Microprocessor Report*, 11(9), 1997.

[VdWMH11] R.F. Van der Wijngaart, T.G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1) :73–83, 2011.

[Ver10] VersionOne. State of agile development, 2010. http ://www.versionone.com/pdf/2011_State_of_Agile_Development_Survey_Results.pdf.

[VM01] A. Van Moorsel. Metrics for the internet age : Quality of experience and quality of business. In *Fifth International Workshop on Performability*

Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany, volume 34, pages 26–31. Citeseer, 2001.

[VOVDLKM00] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[Whi94] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2) :65–85, 1994.

[WSG+12] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goeschka. On patterns for decentralized control in self-adaptive systems. *Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science. Springer*, 2012.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

[ZGC09] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM.

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 31, 40

ADL Architecture Description Language. 38, 39, 47, 55

CPS Cyber Physical Systems. 134, 138–142, 193, 195, 197

DAS Dynamically Adaptive System. 1, 2, 10–14, 33, 34, 36, 39, 41, 42, 44, 45, 52, 56–58, 63, 70, 96, 111, 196

DAUM Dynamic Adaptation Using Models. 19, 20

DDAS Distributed Dynamically Adaptive System. 2, 3, 15, 21, 33, 36, 37, 39, 40, 42–47, 49–53, 55, 56, 58, 60–62, 69, 71–73, 75, 76, 93–95, 98, 100, 101, 104, 105, 108, 111, 114, 119–121, 127, 128, 131, 133–135, 139, 153, 167, 168, 186–189, 193, 195–197, 199–202, 204–206

DSL Domain-Specific Language. 10, 23, 56, 169

EAI Enterprise Application Integration. 29

EIP Enterprise Integration Pattern. 29, 202

ESB Enterprise Service Bus. 29

IDL Langage de Définition d'Interfaces. 24, 41–43

IDM Ingénierie dirigée par les modèles. 2, 9–11, 23, 24, 73, 82, 133, 168, 196, 199

IoS Internet Of Services. 138

IoT Internet Of Things. 14, 138

IPC Inter processus call. 28

JBI Java Business Integration. 35, 36, 84

M@R Model@Runtime. 2, 12, 38, 94–100, 103, 114, 127, 134, 137, 140, 143, 199, 200, 204

M@RC Model@Runtime Core. 105, 106, 108–110, 112–115, 117, 121, 128, 133, 134, 139

MAPE Monitor Analyze Process Execute. 33–37, 59–61, 72, 96, 101, 105, 110, 195

MDA Model Driven Architecture (architecture dirigée par les modèles). 10, 11, 21

MEP Message Exchange Pattern. 81, 82

MOF Meta Object Facility. 23, 24, 78

MOM Message-Oriented Middleware. 27

plasticité capacité à se modeler en fonction des changements d'exigences. 9–11, 13, 195

RPC Remote Procedure Call. 27, 81, 82, 84, 85, 168

SCA Service Component Architecture. 41

SOA Service Oriented Application / Architecture. 27

TRM Tactical Reasoning Method. 19

Table des figures

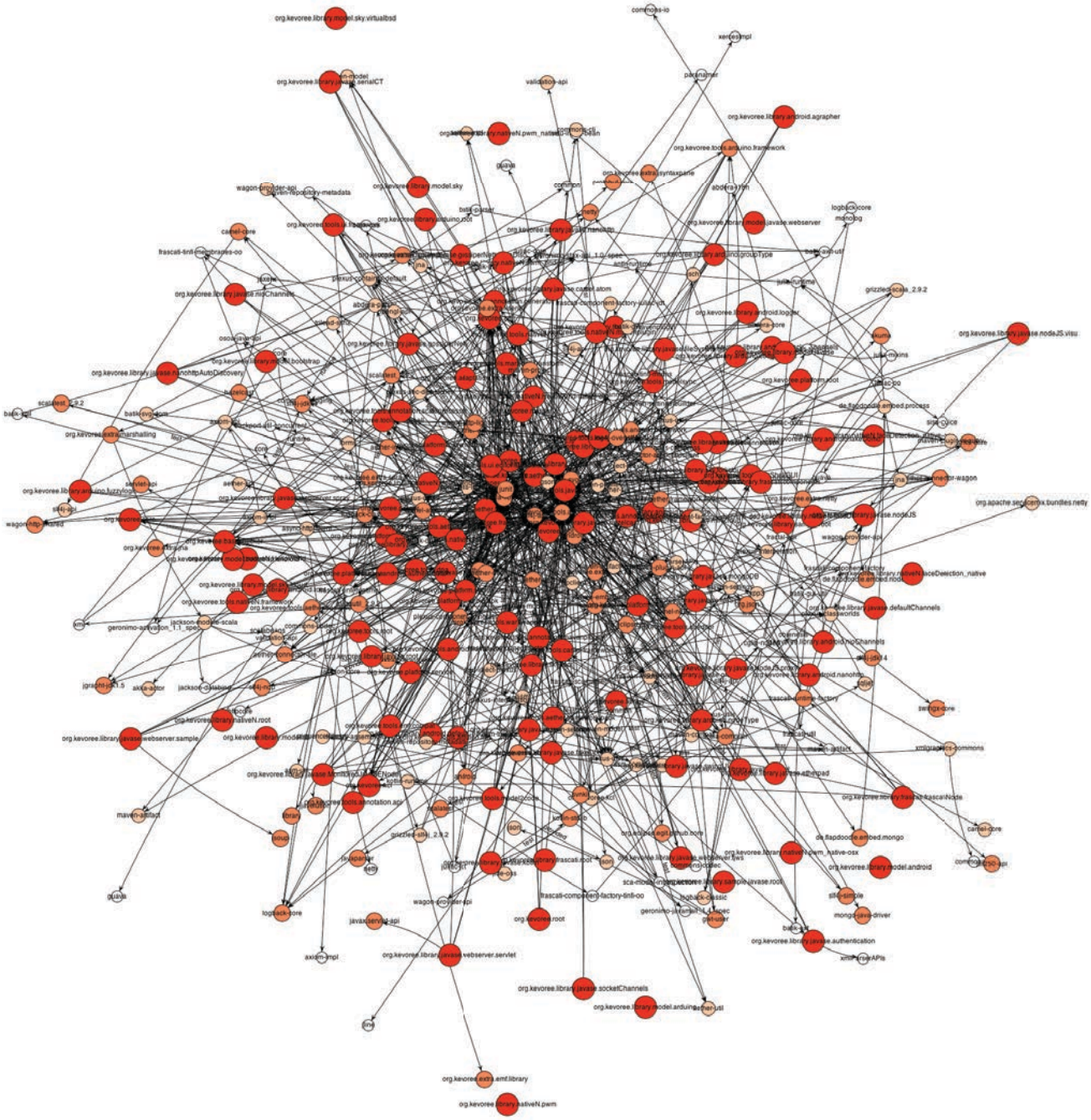
Cinquième partie

Annexe

Listing 1– Algorithme de comparaison des vectorClocks

```
def compare (v1: VectorClock, v2: VectorClock): Occured = {
    /* If one instance is null => priority to none null version */
    if (v1 == null) { return Occured.BEFORE }
    if (v2 == null) {return Occured.AFTER }
    // We do two checks: v1 <= v2 and v2 <= v1 if both are true then
    var largerBigger = false
    var smallerBigger = false
    var largerIsV1 = true
    var larger = 0
    var smaller = 0
    var largerClock: VectorClock = null
    var smallerClock: VectorClock = null
    val sizeEquals = v1.getEntiesCount == v2.getEntiesCount
    if (v1.getEntiesCount >= v2.getEntiesCount) {
      largerClock = v1
      smallerClock = v2
      largerIsV1 = true
    } else {
      largerClock = v2
      smallerClock = v1
      largerIsV1 = false
    }
    for (entry1 <- largerClock.getEntiesList) {
      var check = false
      val ite = smallerClock.getEntiesList.iterator
      while (!check && ite.hasNext) {
        val entry2 = ite.next
        if (entry1.getNodeID.equals(entry2.getNodeID)) {
          if (entry1.getVersion > entry2.getVersion) {
            largerBigger = true
          } else if (entry2.getVersion > entry1.getVersion) {
            smallerBigger = true
          }
          larger = larger + 1
          smaller = smaller + 1
          check = true
        }
      }
    }
    /* Okay, now check for left overs */
    if (larger < largerClock.getEntiesCount) {
      largerBigger = true
    }
    if (smaller < smallerClock.getEntiesCount) {
      smallerBigger = true
    }
    larger match {
      case _ if (!largerBigger && !smallerBigger) => Occured.AFTER
      case _ if (!largerBigger && smallerBigger && sizeEquals) => Occured.BEFORE
      case _ if (!largerBigger && smallerBigger && !sizeEquals) => Occured.CONCURRENTLY
      case _ if (largerBigger && !smallerBigger && largerIsV1) => Occured.AFTER
      case _ if (largerBigger && !smallerBigger && !largerIsV1) => Occured.BEFORE
      case _ if (!largerBigger && smallerBigger && largerIsV1) => Occured.BEFORE
      case _ if (!largerBigger && smallerBigger && !largerIsV1) => Occured.AFTER
      case _ => Occured.CONCURRENTLY
    }
  }
```

FIGURE 1 – Diagramme de dépendance des artefacts Maven du projet Kevoree



VU :
Le Directeur de Thèse
(Nom et Prénom)

VU :
Le Responsable de l'Ecole Doctorale
(Nom et Prénom)

VU pour autorisation de soutenance

Rennes, le

Le président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :
Le Président de Jury,
(Nom et Prénom)

Résumé

La complexité croissante des systèmes d'information modernes a motivé l'apparition de nouveaux paradigmes (objets, composants, services, etc), permettant de mieux appréhender et maîtriser la masse critique de leurs fonctionnalités. Ces systèmes sont construits de façon modulaire et adaptable afin de minimiser les temps d'arrêt dûs aux évolutions ou à la maintenance de ceux-ci. Afin de garantir des propriétés non fonctionnelles (p. ex. maintien du temps de réponse malgré un nombre croissant de requêtes), ces systèmes sont également amenés à être distribués sur différentes ressources de calcul (grilles). Outre l'apport en puissance de calcul, la distribution peut également intervenir pour distribuer une tâche sur des nœuds aux propriétés spécifiques. C'est le cas dans le cas des terminaux mobiles proches des utilisateurs ou encore des objets et capteurs connectés proches physiquement du contexte de mesure. L'adaptation d'un système et de ses ressources nécessite cependant une connaissance de son état courant afin d'adapter son architecture et sa topologie aux nouveaux besoins. Un nouvel état doit ensuite être propagé à l'ensemble des nœuds de calcul. Le maintien de la cohérence et le partage de cet état est rendu particulièrement difficile à cause des connexions sporadiques inhérentes à la distribution, pouvant amener des sous-systèmes à diverger.

En réponse à ces défi scientifiques, cette thèse propose une abstraction de conception et de déploiement pour systèmes distribués dynamiquement adaptables, grâce au principe du Model@Runtime. Cette approche propose la construction d'une couche de réflexion distribuée qui permet la manipulation abstraite de systèmes répartis sur des nœuds hétérogènes. En outre, cette contribution introduit dans la modélisation des systèmes adaptables la notion de cohérence variable, permettant ainsi de capturer la divergence des nœuds de calcul dans leur propre conception. Cette couche de réflexion, désormais cohérente "à terme", permet d'envisager la construction de systèmes adaptatifs hétérogènes, regroupant des nœuds mobiles et embarqués dont la connectivité peut être intermittente. Cette contribution a été concrétisée par un projet nommé "Kevoree" dont la validation démontre l'applicabilité de l'approche proposée pour des cas d'usages aussi hétérogènes qu'un réseau de capteurs ou une flotte de terminaux mobiles.

Abstract

Model@Runtime for continuous development of heterogeneous distributed adaptive systems

The growing complexity of modern IT systems has motivated the development of new paradigms (objects, components, services,...) to better cope with the critical size of their functionalities. Such systems are then built as a modular and dynamically adaptable compositions, allowing them to minimize their down-times while performing evolutions or fixes. In order to ensure non-functional properties (i.e. request latency) such systems are distributed across different computation nodes. Besides the added value in term of computational power (cloud), this distribution can also target nodes with dedicated properties such as mobile nodes and sensors (internet of things), physically close to users for interactions.

Adapting a system requires knowledge about its current state in order to adapt its architecture to its evolving needs. A new state must be then disseminated to other nodes to synchronize them. Maintaining its consistency and sharing this state is a difficult task especially in case of sporadic connexions which lead to divergent state between sub-systems.

To tackle these scientific problems, this thesis proposes an abstraction to design and deploy distributed adaptive systems following the Model@Runtime paradigm. From this abstraction, the proposed approach allows defining a distributed reflexive layer to manipulate heterogeneous distributed nodes. In particular, this contribution introduces variable consistencies in model definition and divergence in system conception. This reflexive layer, eventually consistent allows the construction of distributed adapted systems even on mobile nodes with intermittent connectivity. This work has been realized in an open source project named Kevoree, and validated on various distributed systems ranging from sensor networks to "cloud" computing.